DOCUMENT RESUME

ED 111 347 IR 002 374

AUTHOR	Weyer, S. A.; Cannara, A. B.
TITLE	Children Learning Computer Programming: Experiments with Languages, Curricula and Programmable Devices. Technical Report No. 250.
INSTITUTION	Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	SU-IMSSS-TR-250
PUB DATE	27 Jan 75
NOTE	228p.
EDRS PRICE	MF-$0.76 HC-$12.05 Plus Postage
DESCRIPTORS	Children; *Computer Programs; Computers; Computer Science; *Computer Science Education; *Curriculum Design; Experimental Curriculum; Intermediate Grades; Junior High Schools; Program Descriptions; *Programing Languages
IDENTIFIERS	LOGO; Machine Language; Simper

ABSTRACT

An experiment was conducted to study how children, aged 10-15 years, learn concepts relevant to computer programing and how they learn modern programing languages. The implicit educational goal was to teach thinking strategies through the medium of programing concepts and their applications. The computer languages Simper and Logo were chosen because they are computationally general, relatively easy to learn, interactive with powerful editing features, and are highly dissimilar. The experiment included significant tutoring, curriculum design, and various special output devices such as graphic displays, robots, electric trains, and sound synthesizers. The report is divided into six major sections: (1) introduction: background and motivation; (2) programing facilities; (3) student selection, grouping and tutoring; (4) curricula; (5) data acquisition and analysis; and (6) results. Among the results were suggested modifications to both the Simper and Logo languages and to the curriculum designed to teach them. (KKC)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED111347

CHILDREN LEARNING COMPUTER PROGRAMMING: EXPERIMENTS WITH LANGUAGES, CURRICULA AND PROGRAMMABLE DEVICES

BY

S. A. WEYER AND A. B. CANNARA

TECHNICAL REPORT NO. 250

JANUARY 27, 1975

PSYCHOLOGY AND EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

IR 002 374

TECHNICAL REPORTS

PSYCHOLOGY SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

(Place of publication shown in parentheses; if published title is different from title of Technical Report, this is also shown in parentheses.)

125 W. K. Estes. Reinforcement in human learning. December 20, 1967. (In J. Tapp (Ed.), Reinforcement and behavior. New York: Academic Press, 1969. Pp. 63-94.)

126 G. L. Wolford, D. L. Wessel, and W. K. Estes. Further evidence concerning scanning and sampling assumptions of visual detection models. January 31, 1968. (Perception and Psychophysics, 1968, 3, 439-444.)

127 R. C. Atkinson and R. M. Shiffrin. Some speculations on storage and retrieval processes in long-term memory. February 2, 1968. (Psychological Review, 1969, 76, 179-193.)

128 J. Holmgren. Visual detection with imperfect recognition. March 29, 1968. (Perception and Psychophysics, 1968, 4(4), .)

129 L. B. Mlodnosky. The Frostig and the Bender Gestalt as predictors of reading achievement. April 12, 1968.

130 P. Suppes. Some theoretical models for mathematics learning. April 15, 1968. (Journal of Research and Development in Education, 1967, 1, 5-22.)

131 G. M. Olson. Learning and retention in a continuous recognition task. May 15, 1968. (Journal of Experimental Psychology, 1969, 81, 381-384.)

132 R. N. Hartley. An investigation of list types and cues to facilitate initial reading vocabulary acquisition. May 29, 1968. (Psychonomic Science, 1968, 12(b), 251-252; Effects of list types and cues on the learning of word lists. Reading Research Quarterly, 1970, 6(1), 97-121.)

133 P. Suppes. Stimulus-response theory of finite automata. June 19, 1968. (Journal of Mathematical Psychology, 1969, 6, 327-355.)

134 N. Moler and P. Suppes. Quantifier-free axioms for constructive plane geometry. June 20, 1968. (Compositio Mathematica, 1968, 20, 143-152.)

135 W. K. Estes and D. P. Horst. Latency as a function of number of response alternatives in paired-associate learning. July 1, 1968.

136 M. Schlag-Rey and P. Suppes. High-order dimensions in concept identification. July 2, 1968. (Psychometric Science, 1968, 11, 141-142.)

137 R. M. Shiffrin. Search and retrieval processes in long-term memory. August 15, 1968.

138 R. D. Freund, G. R. Loftus, and R. C. Atkinson. Applications of multiprocess models for memory to continuous recognition tasks. December 18, 1968. (Journal of Mathematical Psychology, 1969, 6, 576-594.)

139 R. C. Atkinson. Information delay in human learning. December 18, 1968. (Journal of Verbal Learning and Verbal Behavior, 1969, 8, 507-511.)

140 R. C. Atkinson, J. E. Holmgren, and J. F. Juola. Processing time as influenced by the number of elements in the visual display. March 14, 1969. (Perception and Psychophysics, 1969, 6, 321-326.)

141 P. Suppes, E. F. Loftus, and M. Jerman. Problem-solving on a computer-based teletype. March 25, 1969. (Educational Studies in Mathematics, 1969, 2, 1-15.)

142 P. Suppes and M. Morningstar. Evaluation of three computer-assisted instruction programs. May 2, 1969. (Computer-assisted instruction. Science, 1969, 166, 343-350.)

143 P. Suppes. On the problems of using mathematics in the development of the social sciences. May 12, 1969. (In Mathematics in the social sciences in Australia. Canberra: Australian Government Publishing Service, 1972. Pp. 3-15.)

144 Z. Domotor. Probabilistic relational structures and their applications. May 14, 1969.

145 R. C. Atkinson and T. D. Wickens. Human memory and the concept of reinforcement. May 20, 1969. (In R. Glazer (Ed.), The nature of reinforcement. New York: Academic Press, 1971. Pp. 66-120.)

146 R. J. Titiev. Some model-theoretic results in measurement theory. May 22, 1969. (Measurement structures in classes that are not universally axiomatizable. Journal of Mathematical Psychology, 1972, 9, 200-205.)

147 P. Suppes. Measurement: Problems of theory and application. June 12, 1969. (In Mathematics in the social sciences in Australia. Canberra: Australian Government Publishing Service, 1972. Pp. 613-622.)

148 P. Suppes and C. Ihrke. Accelerated program in elementary-school mathematics--The fourth year. August 7, 1969. (Psychology in the Schools, 1970, 7, 111-126.)

149 D. Rundus and R. C. Atkinson. Rehearsal processes in free recall: A procedure for direct observation. August 12, 1969. (Journal of Verbal Learning and Verbal Behavior, 1970, 9, 99-105.)

150 P. Suppes and S. Feldman. Young children's comprehension of logical connectives. October 15, 1969. (Journal of Experimental Child Psychology, 1971, 12, 304-317.)

151 J. H. Laubsch. An adaptive teaching system for optimal item allocation. November 14, 1969.

152 R. L. Klatzky and R. C. Atkinson. Memory scans based on alternative test stimulus representations. November 25, 1969. (Perception and Psychophysics, 1970, 8, 113-117.)

153 J. E. Holmgren. Response latency as an indicant of information processing in visual search tasks. March 16, 1970.

154 P. Suppes. Probabilistic grammars for natural languages. May 15, 1970. (Synthese, 1970, 11, 111-222.)

155 E. M. Gammon. A syntactical analysis of some first-grade readers. June 22, 1970.

156 K. N. Wexler. An automaton analysis of the learning of a miniature system of Japanese. July 24, 1970.

157 R. C. Atkinson and J. A. Paulson. An approach to the psychology of instruction. August 14, 1970. (Psychological Bulletin, 1972, 78, 49-61.)

158 R. C. Atkinson, J. D. Fletcher, H. C. Chetin, and C. M. Stauffer. Instruction in initial reading under computer control: The Stanford project. August 13, 1970. (In A. Romano and S. Rossi (Eds.), Computers in education. Bari, Italy: Adriatica Editrice, 1971. Pp. 69-99. Republished: Educational Technology Publications, Number 20 in a series, Englewood Cliffs, N. J.)

159 D. J. Rundus. An analysis of rehearsal processes in free recall. August 21, 1970. (Analyses of rehearsal processes in free recall. Journal of Experimental Psychology, 1971, 89, 63-77.)

160 R. L. Klatzky, J. F. Juola, and R. C. Atkinson. Test stimulus representation and experimental context effects in memory scanning. (Journal of Experimental Psychology, 1971, 87, 281-288.)

161 W. A. Rottmayer. A formal theory of perception. November 13, 1970.

162 E. J. F. Loftus. An analysis of the structural variables that determine problem-solving difficulty on a computer-based teletype. December 18, 1970.

163 J. A. Van Campen. Towards the automatic generation of programmed foreign-language instructional materials. January 11, 1971.

164 J. Friend and R. C. Atkinson. Computer-assisted instruction in programming: AID. January 25, 1971.

CHILDREN LEARNING COMPUTER PROGRAMMING: EXPERIMENTS WITH LANGUAGES, CURRICULA AND PROGRAMMABLE DEVICES

by

S. A. Weyer and A. B. Cannara

TECHNICAL REPORT NO. 250

January 27, 1975

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

PSYCHOLOGY AND EDUCATION SERIES

Reproduction in Whole or in Part Is Permitted
for Any Purpose of the United States Government

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

Table of Contents

Index to Tables iv

Index to Figures v

Sections

1 Introduction: Background and Motivation 1

2 Programming Facilities 10

  2.1 Languages 11

    2.1.1 Command Parsing and Execution 15

    2.1.2 Editing and Debugging Facilities 28

  2.2 Peripheral Devices 42

    2.2.1 Standard Alphanumeric Terminals 44

    2.2.2 Vector Graphics Terminals 46

    2.2.3 Output Devices: Plotter, Turtle, Train and Audio . . 52

3 Student Selection, Grouping and Tutoring 57

4 Curricula 71

  4.1 Logo 75

  4.2 Simper 78

  4.3 Contrasts 81

5 Data Acquisition and Analysis 87

6 Results 106

6.1 Understanding the Students 115
6.2 Evaluation of Simper and Logo 135
6.3 Implications for Language and Curriculum Design 142

7 Final Comments 145

Appendices 146

1 Graphics 147
1.1 TEC(R) 148
1.2 IMLAC(R) 151

2 Controllable Devices 160
2.1 Robot Turtle and Music Box 160
2.2 Train 163

3 Details Pertinent to the Preliminary Test 167
3.1 An Example of Commercial Evaluation 167
3.2 Sample Problems from the Preliminary Test 169

4 Sample Logo Curriculum 172

5 Sample Simper Curriculum 195

Acknowledgements 216

References 217

Index to Tables

I Some Fundamental Programming Concepts 9

II Simper Machine Operations 12

III Some Logo Primitives 14

IV Simper Interpreter Commands 20

V Simper/Logo Line-editing Commands 29

VI Logo's Procedure Editing and Debugging Commands 35

VII Logo's File-manipulation Commands 38

VIII IMSSS Logo/TEC(R) Commands 45

IX IMSSS Logo Turtle-Graphics Commands 48

X IMSSS Logo Animation Commands 49

XI IMSSS Logo Train Commands 55

XII Experimental Groups 58

XIII Discussions of the Concepts in the Curricula 73

Index to Figures

1. Simper and Logo Sample Dialogues 16
2. Structure of Simper's Simulated Machine 17
3. Schematic of Logo's Memory Space 25
4. An Example of Logo's Use of Its Execution Stack 27
5. Supporting Structure for the Simper Assembler 30
6. Two Uses of Simper's 'SLIDE' Command 32
7. Displaying a Simper Program's Activity 39
8. Tracing a Logo Procedure's Activity 41
9. Programming System Structure 43
10. Successive Frames from a Logo-Animation "Movie" 51
11. Schematic of the Logo-Controlled Train Layout 54
12. Some Information Characterizing the Students 60
13. Some "Wrong" Answers from the Preliminary Test 66
14. Some Novel Answers from the Preliminary Test 67
15. Student Ranking on the Preliminary Test 69
16. A Simple Quantitative Analysis of Protocols 99
17. Student Preferences 107
18. Breakdown of Students' Programming Time 109
19. Simper Students' Performance Versus Pretest Rank 112
20. Logo Students' Performance Versus Pretest Rank 114

1 Introduction: Background and Motivation

In this report, we discuss in detail an experiment which took place at the Institute for Mathematical Studies in the Social Sciences (IMSSS) during the summer of 1973. A brief outline has appeared elsewhere (Cannara & Weyer, 1974). We also discuss related, informal events, which derived from and occurred subsequent to the experiment.

The experiment attempted to study how children learn: (a) concepts relevant to computer programming, and (b) modern programming languages. We will discuss the languages used, why they were chosen and what the experiment suggested in terms of the design of the languages as well as programming languages in general. Because the particular concepts and languages were to be taught to naive programmers, the experiment included a significant tutoring and curriculum design project. This phase of our work is carefully detailed. We include discussion of several special output devices which the children controlled via their programs in order to draw lines on paper, animate pictures on graphic displays, move a robot, control an electric train, and synthesize sound. We examine these devices in terms of their motivational value to children, and how and to what extent they might offer means for posing pedagogically useful problems to student programmers.

The language and curriculum design aspects of this experiment were partly intended to lay groundwork for a subsequent, more refined study of children's interactions with programming. Results of that experiment and some further analyses of data from this experiment will appear in a later report (Cannara, 1975).

It is not a new idea that children can and should learn how to program a computer, so that they too might access its unparalleled power as a tool for thinking. Various computer scientists have worked to cast the computer as a personal "mathematical laboratory" (Brown, Dwyer, Feurzeig, Kay, Papert). In 1965, Feurzeig proposed that a suitably programmed machine could create a constructively interactive environment with the potential to enhance a child's interest and learning in mathematics. Since then, attempts have been made to realize such mathematical laboratories in contexts ranging from formal logic (Goldberg, 1973) and calculus (Kimball, 1973) to computer programming or "mathematizing" (Feurzeig, Papert, Bloom and Solomon, 1969). In such environments, students may enjoy broad freedom to explore, interactively and constructively, disciplines which are frequently deprived of substance by either the classroom lecture or traditional computer-assisted instruction (CAI).*

In any computerized implementation of a mathematical laboratory, a program simulates the system of interest; the student communicates with this simulation via a formal language. The semantics of that language access the constructive abilities of the laboratory, the syntax is just a new set of notational conventions. Both must be considered carefully by the laboratory's designer and both must be mastered by the student.

---

* The reader might examine Ellis (1974) or Oettinger and Marks (1969), especially Ellis' nontechnical critique of present applications of computers in education.

Of all possible mathematical laboratories, the most general are those which give students full computational access to a computer, by allowing them to write programs. The means for communicating with such laboratories are programming languages, which define tools available to anyone using the laboratories to formalize ideas. The formalization of ideas is a fundamental aspect of mathematics. If, by a free interpretation of Church's thesis*, any ideas which may be formalized may be studied concretely via a computer program, then, by learning programming in full generality, students can learn how to construct laboratories to study any ideas they wish to think about. Furthermore, because programming offers a way of formalizing thoughts to produce concrete effects, students can learn something about thinking. For this reason, we and others believe that the natural place for the computer is in the schools, where thinking and "thinking about thinking" (a notion promulgated by Papert) can and should be taught.

From an educator's viewpoint, the theory and practice of computation offer much: (a) the formalization of ideas as sequences of instructions, (b) methods for modelling real-world processes, and (c) metaphors for describing machine and human information processing. These form a nucleus of thinking techniques which expose what Papert has termed "powerful ideas". Concepts of programming and thinking can be taught as natural and inseparable partners, with emphasis on improving students' abilities to scrutinize their own thinking about the world.

---

* For discussions of this important conjecture, see Manna (1972) or Minsky (1967).

The computer's ability to simulate responds to the ingenuities of students (for example, see the work of Papert, 1970 or Brown and Rubinstein, 1973) with the same spectacular generality it has provided to professional researchers (good examples are in Levison, Ward and Webb, 1973; in Toomre, 1973 and in Winograd, 1971). More recently, as computing machinery has become cheaper and more accessible, it has begun to pervade the high schools. It seems reasonable that this trend should soon extend interactive computation into the elementary schools.

The foregoing remarks were meant to justify our desire to study programming as an intellectual activity for children and programming languages as tools for such activity. If access to interactive computation will soon become commonplace for vast numbers of children, at school or at home, then we certainly should be trying now to understand how to make the most fruitful use of the technology. As a medium for manipulating and expressing ideas, the personally accessible computer may stand well above everything since the printing press.* It is important, therefore, to study the computer and children (or adults) as tool and users so that the tool may be honed to maximum usefulness (recreational, artistic or educational).

Because it is widely believed that young children can benefit intellectually by learning programming, numerous research projects have been set up to teach particular programming languages (e.g. Feurzeig and Lukas 1972a; Fischer, 1973; Folk, Statz and Seidman, 1974; Milner,

---

* See Kay (1972a, 1972b) or Brand (1974, pp. 64-71) for one view of the near future of computing.

1973 or Roman, 1972). However, apparently none has attempted to make explicit the broad range of programming concepts and their relationship to a student's world of thought. In such terms, many projects have pursued hazy and sometimes arbitrary goals that concentrated on teaching an available language through ad hoc problem-solving situations. Little effort has been expended on generalizing those situations and the solution strategies used. A study by Folk, et al., (1974) is perhaps the most extensive attempt to specify relationships between programming concepts and the development of children's thinking processes. But their analysis is confined to classical analysis-of-variance models and the concomitant testing of rather broad hypotheses virtually ignores a wealth of valuable detail in student protocols -- the type of data we value most.

Teaching programming is a tutorial endeavor. A programming tutor must be ready to intelligently suggest, accept and comment on an arbitrarily wide range of student interactions and program synthesis. In any tutorial atmosphere, the details of errors made by a student are extremely useful. They do more than indicate what the student does not understand, they indicate how the student views the problem at hand in terms of his or her own view of the world. Extending a suggestion of Papert's, if a student responds to a posed problem at all, that response is typically correct by the student's personal analysis. So the student is surprised to hear "wrong". It is the tutor's responsibility to try to divine the reasons for the student's error. This frequently means that the tutor must act as does a detective attempting to elicit evidence from someone from a foreign land. Much of the subsequent interaction must be devoted to laying a common foundation of terms -- their definitions and relations. The tutor necessarily learns something about the student's world view and is better prepared to handle future errors and future students.

Errors are not "bad". They provide valuable feedback to be exploited for a student's benefit. Because students sense this and respond positively, much of what is presently considered to be advanced research in computer-assisted instruction concentrates on establishing such a close relationship between tutor (albeit mechanical) and student.

The construction of programs which can tutor humans with human proficiency has been the goal of many researchers. No one has been fully successful yet, because the fundamental activities of a good tutor are tied irrevocably to humanness of language and knowledge. The theoretical power of a computer may be sufficient to simulate human intellect, but we do not understand ourselves well enough to communicate even a coarse description of our intellect to any recipient. Those who have recognized the nature of this problem have come closest to success in limited contexts (e.g. Brown and Burton, 1974; Carbonell, 1970 or Winograd, 1971). Teaching programming is perhaps the most general tutorial activity one could care to mechanize, so we believe that detailed studies of students learning to program can help to characterize tutorial interactions in general.

Any tutor must (a) understand the subject being taught and (b) possess a strategy for handling errors that is adaptable to the demands set by individual students. Unfortunately, the bulk of past efforts in

CAI have bypassed (a) and have sought to discover techniques for manipulating student performance (most frequently measured in ways more convenient for the researcher than beneficial to the student) by attacking (b) in narrow contexts (e.g. the reader should critically examine Smallwood, 1962 or the examples used by Suppes in Wittrock, 1973). The result too often has been a simple transfer of programmed instruction from paper or film to computer storage, applying very little, from the student's vantage, of the computer's computational potential.* Largely in conjunction with advances in artificial-intelligence research, (a) and (b) have been attacked together (e.g. Goldberg, 1973, Brown and Burton, 1974, Kimball, 1973).

However, any general tutorial system for teaching programming is destined to occasionally fail the student; because of its generality, it must occasionally tackle unsolvable (uncomputable) problems.** In other words, it must pass judgment on the correctness of a student's programs, and we know that there exists no general procedure for deciding that an arbitrary program is correct or incorrect. But a human tutor is faced with the same situation, and the range of solvable problems is so broad that this hard theoretical fact has discouraged neither researchers nor teachers. "Proof of program correctness" and "automatic program synthesis" are active topics in computational research which have clear bearing on future success in constructing competent computer-based tutorial systems. The work to be reported here attempts to characterize some of the situations that human and mechanical tutors for programming will confront and must be prepared to resolve.

---

* We agree with Dwyer (1972) who has said that CAI fails in "reproducing the excitement of masterful teaching". We would add that only rarely have CAI workers even attempted to capture masterful teaching.

**Discussions of the uncomputable appear in Davis (1965) and in Minsky (1967).

Our implicit educational goal is teaching thinking strategies by teaching programming concepts and their applications. Ideally, a student should look to his or her own life experience for applications of the tools which an understanding of the concepts supplies. This, we believe, is the ultimate justification for teaching programming. For programming to succeed (from the students' point of view) as part of any educational experience, we must be concerned with each student's individual approach to it. The power of a programming laboratory derives from the fact that students do more than interact with it, they intervene. Through its language they formulate and activate ideas and, in doing so, mould the laboratory to their own purposes. From primitive tools available to them at the start, they derive new ones, and from these, others, ad infinitum.

That programming concepts provide an invaluable link between formalized thinking and perceived reality is certainly not a new axiom (Berry, 1964). It was assumed, perhaps tacitly, in much of the research quoted here. However, no study has attempted to teach a full range of relevant concepts from computation theory (see Table I) which we believe is essential to establishing that link. Another motivation for our work has been a desire to contrast programming languages and how

Table I

Some Fundamental Programming Concepts

1. Machine as a tool manipulated with a command language
2. Machine possessing an alterable memory
3. Literal expressions
4. Name-value associations
5. Evaluation and symbol-substitution
6. Execution of stored programs
7. Programs which make decisions
8. Procedures (algorithms)
9. Evaluation of arguments to procedures
10. Procedures as realizations of functions (transformations)
11. Composition of functions
12. Partial and total functions
13. Computational context (local versus global environments)
14. Evaluation in changing environments
15. Induction (recursion and iteration)
16. Data structures as defined by functions
17. Problem formulation (representation)
18. Incomplete algorithms (heuristics)

they aid or hinder acquisition of the set of concepts we have said we value. Syntactic differences among languages are of but incidental interest. Most important is how the meanings (semantics) of a language, accessible via its grammatical rules (syntax) and defined on the structure of some underlying machine, can illuminate the concepts. Furthermore, we feel there is a need to investigate the educational value of some of the many types of devices that may be used by students and controlled by their programs.

So, the problem we posed for research can be summarized in two questions: (a) How do the characteristics of programming languages and devices influence a child's motivation and ability to learn programming concepts and apply them to the solution of problems? and (b) How do children relate programming concepts with their real-life experiences?

2 Programming Facilities

Our tutorial structure attempted to impart an understanding of the concepts in Table I and fluency in two, very different programming languages. This required the development of (a) interactive laboratories (interpreters) for the languages and devices used, (b) parallel curricula for teaching the concepts, (c) means for acquiring data on each student's interactions, and (d) means for assessing each student's aptitude for programming and mastery of the concepts.

Part of requirement (a) was easily met by using existing interpreters for two languages, Logo and Simper, developed specifically to teach children computer programming. The development of some of the devices used and requirements (b), (c) and (d) defined the work to be done preliminary to the actual experiment.

2.1 Languages

The languages Simper and Logo were chosen because they are computationally general, they are relatively easy to learn, they are interactive with powerful editing features, and they are highly dissimilar.

Simper was developed by Lorton and Slimick (1969) at IMSSS as a simple simulation of an imaginary machine resembling an Hewlett-Packard model 2000. It was used to teach business applications of programming to students at Woodrow Wilson High School in San Francisco via remote lines from the IMSSS PDP-1. Simper was implemented later on that high school's HP-2000F in Basic. At IMSSS, it has been expanded and rewritten in the Algol-60 subset of Sail (Swinehart and Sproull, 1971) by the authors.

Simper, like Logo, is designed for interactive use. It is an assembly language interpreter for a simple decimal machine with an addressable program counter. Its instruction set typifies those of early minicomputers and is similar to, but simpler than, that of the language Mix (Knuth, 1970). As a programming laboratory, Simper has three functional components: (1) an interpreter which handles editing and general management of programs, (2) a real-time assembler which translates symbols and mnemonic instructions (listed in Table II) into machine language, and (3) a simulator for the underlying machine. This

Table II

Simper Machine Operations

Mnemonic	Action (if not obvious)
PUT	value of address field to register
LOAD	copy value in addressed cell into register
STORE	inverse of LOAD
ADD	add value in addressed cell to register
SUBTRACT	
MULTIPLY	
DIVIDE	
LAND	decimal digit-wise minimum between register and memory
LOR	decimal digit-wise maximum
LEXOR	"exclusive or": LOR except for equal digits
JUMP	transfer to address if register is non-zero
JASK	transfer to address if a key has been typed
COMPARE	three-way skip on memory cell's value greater than, equal to, or less than register's value
SHIFT	
ROTATE	
EXCHANGE	flip contents of two registers
INCREMENT	
NEGATE	
ERROR	overflow error code to register
ASK	decimal numeral from keyboard to register
WRITE	inverse of ASK
CASK	ASCII character from keyboard to register
CWRITE	inverse of CASK
IOT	input/output transfer (for graphics etc.)
RANDOM	random 10-digit integer to register
TIME	seconds since midnight to register
WAIT	defer execution for milliseconds in register
HALT	stop execution
NOP	no-operation

Typical Instructions

```
ASK B
ADD B 100
HALT
```

Each instruction may have one, two or three parts, (1) the operation, (2) the register and (3) the address.

system allows students to generate and easily "debug" nontrivial machine-language programs.

Logo (Feurzeig, et al., 1969) is a procedural language whose basic data structures are strings of letters or words. It too was developed for children and has been used extensively in educational research.

The Logo instruction set is easily expanded via procedure (command) definitions, which may be expressed recursively. Commands which a student defines are syntactically equivalent to Logo's primitives. Logo contains essentials of the currently popular Basic language as a subset, but is superior to Basic in terms of mathematical consistency, and clarity of phrasing and control. In addition, Logo begins to address the important question of language extensibility, which we feel is a fundamental measure of the usefulness people can attribute to any language for computing or thinking. Our Logo interpreter was obtained from Bolt, Beranek & Newman Inc. (BBN) of Boston. It is written in Macro assembly language for the PDP-10. For the purposes of our experiment, we modified Logo to communicate with special alphanumeric displays, a model train, an "X-Y" plotter, graphic display terminals and the IMSSS digitized-audio system. During the experiment, several children had access to each of these devices. As a result of obvious student enthusiasm during the main portion of the experiment, Simper was later also modified to access the graphics devices. A partial list of IMSSS Logo's primitive commands appears in Table III.

In the body of this text, we use paired, single quotes (') to denote phrases in the Logo and Simper languages

Table III

Some Logo Primitives (* means peculiar to IMSSS)

Name	Action
TO	allows creation of a new operation (a procedure)
RETURN* or OUTPUT	allows operations to return values to the evaluator
EDIT	allows the user to change an operation's definition
MAKE	associates a name with a value
VALUE* or THING	accesses the value associated with a name
FRONT*	moves the "turtle" or train forward
WHERE*	returns the present location of the train
PLOT*	sends turtle drawing to X-Y plotter or robot
SAY*	causes the audio system to speak a message
PRINT	causes the user's terminal to type a message
REQUEST	asks the user for a message
SNAP*	makes a "snapshot" of graphics picture being drawn
MOVESNAP*	moves a snapshot as part of an animated display
WORD	combines two sets of letters or numbers into one
SENTENCE	combines two words or sentences into a sentence
FIRST	returns the first letter or word in a value
RANDOM	picks a digit between 0 and 9
SAMEP* or IS	are two words or sentences identical?
EQUALP	are two numbers equal?
IF THEN ELSE	decision making

Typical Compound Commands

```
PRINT SENTENCE "THE TRAIN IS AT:" WHERE
FRONT RANDOM
IF EQUALP RANDOM REQUEST THEN SAY "GOOD GUESS" ELSE SAY "OOPS"
```

The disparate natures of Logo and Simper are demonstrated by two sample dialogues (Figure 1) which produce alternative programs for the repeated printing of a keyboard character supplied by the typist. In the figure, prompts from Simper are the current memory address (a decimal numeral) and a ":" or an "!", depending on whether the addressed location is empty or used. Logo prompts "_" at the outer level and "@" at the editing level. A "↑G" indicates a control character typed by the user to stop a potentially endless execution sequence.

Many readers may not be familiar with Logo and most will not be familiar with Simper. The next two sections are intended to fill such gaps. An appreciation of both languages should naturally grow as we discuss the curricula, student data and results later on. The importance of powerful editing and debugging features in Logo and Simper should become especially apparent. As part of the analysis of student interactions, we will discuss changes we have made, or would like to make, to Logo and Simper. A few changes are evident in Tables II and III, which show the states of Simper and Logo after the experiment (e.g. after the addition of graphics capability to Simper via the 'IOT' operation, and new, or alternate command names, such as 'RETURN', in Logo).

2.1.1 Command Parsing and Execution

As outlined above, the Simper interpreter allows its user three basic abilities: (1) entry of machine-language instructions, (2) entry of assembly-language mnemonics and symbols, and (3) various editing and

SIMPER

```
001 :PUT A 43
002 :NAME REPEAT
002 !CWRITE A
003 :PUT P REPEAT
004 :RUN

EXECUTING 1 TO 500
+++++++++++↑G
...23 INSTRS IN .043 SEC.

004 :EDIT 1
001 !CASK A
004 :SLIDE 2:7
002 :ASK B
003 :NEGATE B
004 :JUMP B .+
005 :HALT
006 :INCREMENT B
007 !NAME 4 REPEAT
SWITCHING REPEAT'S REFERENCES
007 !RUN

EXECUTING 1 TO 500
+10
++++++++++
HALT...45 INSTRS IN .117 SEC.

007 !LIST

YOUR PROGRAM:

001 :CAS A
002 :ASK B
003 :NEG B
004 :JUM B .+2 (REPEAT)
005 :HAL
006 :INC B
007 :CWR A
008 :PUT P REPEAT
```

LOGO

```
_TO REPEAT :LETTER:
@10 TYPE :LETTER:
@20 REPEAT :LETTER:
@END
REPEAT DEFINED

_REPEAT "+"
++++++++++↑G
I WAS AT LINE 10 IN REPEAT

_EDIT REPEAT
@EDIT TITLE
@TITLE TO REPEAT :LETTER: :TIMES:
@5 TEST LESSP :TIMES: 1
@7 IFTRUE DONE
@EDIT LINE 20
20 REPEAT :LETTER: DIFFERENCE :TIMES: 1
@END
REPEAT DEFINED

_REPEAT "+" 10
++++++++++_EDIT REPEAT
@6 IFTRUE SKIP
@END
REPEAT DEFINED

_REPEAT "+" 10
++++++++++
_LIST REPEAT

TO REPEAT :LETTER: :TIMES:
5 TEST LESSP :TIMES: 1
6 IFTRUE SKIP
7 IFTRUE DONE
10 TYPE :LETTER:
20 REPEAT :LETTER: DIFFERENCE :TIMES: 1
```

Fig. 1. Simper and Logo Sample Dialogues.

other commands for program management. For category (3), only the syntax of commands will be discussed here; details appear in the next section. One can imagine that, when the Simper interpreter is not running a user's program, it is simply waiting for a message from the user which either falls into one of the three categories above or is unintelligible.

The underlying machine simulated within the Simper interpreter operates on decimal numerals (words), some of which it "understands" as legal instructions. The size and number of memory and register words is adjustable whenever the interpreter is compiled. The machine's organization was as shown in Figure 2. Each of the 250 memory cells

```
                     Registers              Memory Cells

          (10 max.) B: ..........       001: ..........

                    A: ..........       002: ..........
                                          :
(program counter)   P: ..........         :
                                          :
                                        250: .......... (511 max.)


                   Instruction Format

        seven digits: ..   .   ....
                       o    r    a
                        p    e    d
                         e    g    d  --(indirect flag & address)
                          r    i    r
                           a    s    e
                            t    t    s
                             i    e    s
                              o    r
                               n
```

Fig. 2. Structure of Simper's Simulated Machine

and each register could contain a ten-digit decimal numeral. The program counter is simply the "P" register, whose content is usually the memory address of the next instruction to be obeyed. That content can be changed by any instruction which chooses to write into P. Instructions are seven-digits in length and are partitioned (see Figure 2) into three fields: operation, register, and indirect address flag and address.

Each operation mnemonic in Table II has a corresponding two-digit code, each register has a one-digit code. The four-digit field used for addressing may be filled in various legal ways, depending upon the operation to be obeyed. For some operations, the register and/or address fields are not used and can be filled out with zeros. A user may type any legal, seven-digit instruction numeral to the interpreter and it will be stored in the memory location whose address appeared in the interpreter's prompt (e.g. to the left of the ":" in Figure 1). In fact, any numeral, up to ten-digits in length, can be entered into memory this way. Whenever such a message is stored, the next prompt given the user will refer to the next available memory cell.

Assembly language instructions are translated (by a "real-time" assembler) into machine language numerals which are, in turn, stored in the prompted memory cell. The three fields of the target numeral are synthesized from three spaced fields of mnemonics or numerals typed by the user. In this way, machine and assembly language may be mixed within one typed instruction. Translation of instructions which contain no symbols (no names such as 'REPEAT' in Figure 1) is direct.

Symbolic addresses are looked up in a symbol table which is up-dated each time the 'NAME' command is used. Because real-time assembly must account for possible editing changes, a symbol may be used in an address field before 'NAME' has identified it with a cell. In such a case, the address field of the machine instruction is zero and another table, with an entry for each memory cell, is marked to show that the instruction must have its address field "fixed up" if the symbol ever becomes attached to some cell. This additional table, parallel to memory, is also used to hold information on relative addresses, comments entered with instructions, and which cells are named. The need for such extra storage, invisible to the user, will become clearer when Simper's editing features are discussed in the next section.

Program management is handled by a set of immediately executable commands (Table IV). These cannot be executed from within a user's program, and so may be considered a separate language. Syntactically, however, they are similar to assembler instructions. They may have one to three fields (e.g 'LIST', 'DUMP 4:DO+1', and 'NAME 4 REPEAT'), the latter two of which are subject to assembler addressing syntax (plus the range operator ":", read as "to").

Execution of Simper programs is initiated with the 'RUN' command or continued with 'GO'. The value in register P is always used as the address from which to fetch the next instruction, and it is incremented by one just before that instruction is executed. Errors detected at this time are: (1) an illegal program-counter value, (2) an attempt to execute a noninstruction, or (3) a zero address resulting from failure

Table IV

Simper Interpreter Commands

Name	Action
DUMP	display decimal content of memory and registers (symbols too)
LIST or DEBUG	display memory content in assembly language (and machine language, DEBUG shows "secret" tables)
RUN	execute part or all of a program (and display registers)
GO	continue execution (and display registers)
EDIT or FIX	change the content of one or more memory cells (and show prior content)
SLIDE	relocate part or all of a program in memory
SCRATCH	erase part or all of a program
NAME	attach a symbol to a memory cell (and say how much room remains for symbols)
FORGET	erase a symbol
NAMES	list all symbols and their associations (and their values)
SAVE	write memory onto long-term storage
GET	inverse of SAVE
FIELDS	allow abbreviated instructions
NEWS	obtain the latest system news
HELP	obtain general information about Simper
GOODBYE	log out
control-G	stop any activity

Parenthesized phrases describe options explained in the text.

to use 'NAME' to bind a symbol to a cell. If an error message is generated, execution is stopped. During execution, other kinds of errors, such as overflow, may occur which may or may not cause a halt. If there is an error halt or a user interruption, P's value is saved. The user can edit the program and then type 'GO' to continue execution.

The effect of executing an individual instruction may be a change in the values in registers or in memory cells, but not in both types of storage. 'STORE' and 'LOAD' copy values nondestructively in opposite directions between registers and memory cells; 'EXCHANGE' flips the values in two registers; 'IOT' may copy or change more than one memory cell; and 'NOP' does nothing. For arithmetic and logical operations ('ADD' through 'LEXOR' in Table II), results of a computation are always left in the register mentioned in the instruction's register field. 'DIVIDE' is a special case because it computes both a quotient and a remainder, leaving them respectively in the mentioned register and the one adjacent to it.

Logo's interactive structure is more nearly unitary. Its basic piece of executable code is a line composed of one or more commands, and its basic piece of program or procedure (operation definition), is a series of lines. The Logo interpreter is always executing (or capable of executing) a user's commands, which may call upon Logo primitives or the user's own procedures. Control returns to the user only when his or her last command and any commands it might have called have terminated naturally or been aborted. A few of Logo's primitives may not be executed directly by a user's procedure, but there is not a strict distinction between two sets of commands as exists in Simper. However, a quirk in Logo's evaluation scheme imposes a different syntax on editing and management commands versus other primitives and user procedures. We will discuss this later when we take up questions of language design.

A command to the Logo interpreter consists of two parts: (1) the command name (operation) and (2) an argument list, (e.g. 'PRINT "HELLO"'). The appropriate number of arguments must appear after each primitive operation or user-procedure name in any syntactically legal command. Thus, Logo is inherently a prefix language.* Evaluation of non-editing commands is fully general: arguments may be supplied by constants, variables or executable commands (see the bottom of Table III for some examples).

The basic data structure in Logo is the character string. This is broken into two subclasses: words and sentences. A Logo sentence is any string containing words separated by spaces. A Logo word is any string of letters, digits and punctuation. Numerals are simply a subclass of words. Although Logo's arithmetic operations are restricted to integers, arguments may be of arbitrary length (i.e. unlimited magnitude).

There are three ways to access data in Logo: (1) as constants, (2) as values returned by executed commands, and (3) as values associated with names. Constants (literals) are either quoted strings or signed numerals. A quoted string may be a word or a sentence. Quoted numerals are treated as if they were unquoted. Any value that may be expressed as some combination of constants, named values, or values returned by commands may itself be returned by a command, or be associated with some name (see Table III and Figure 1 for some examples). A value is associated with a name by instantiation of a procedure's argument, or by the 'MAKE' operation (e.g. 'MAKE "CAT" "MEOW"'). It is referenced by the 'THING' or 'VALUE' operation or by surrounding the name with colons (e.g. the two commands: 'PRINT VALUE "CAT"' and 'PRINT :CAT:' would each produce "MEOW"). Values may also be used as names, allowing any depth of indirect addressing (e.g. executing 'MAKE :CAT: "NOISE"' would allow 'PRINT :MEOW:' to produce "NOISE").

---

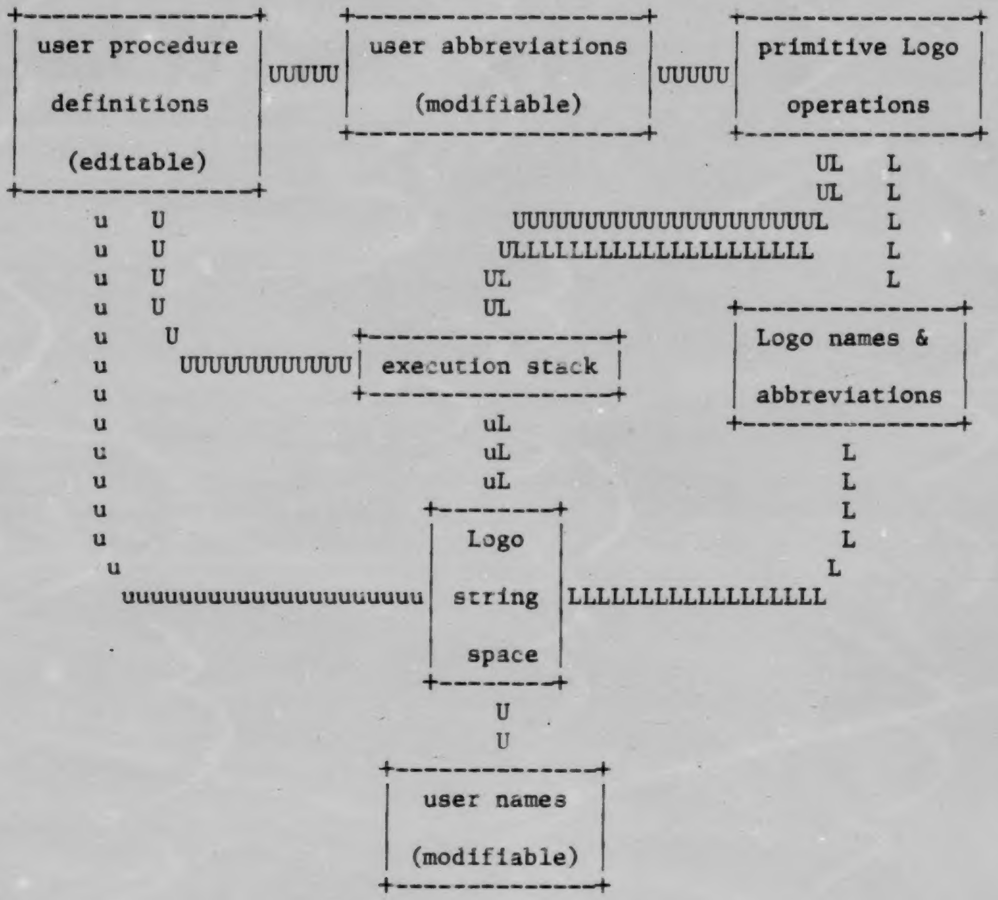
* The version of Logo used in this experiment also allows infix notation for arithmetic operations like: '3+4', but we felt this to be an inconsistent feature and disabled it, e.g. to allow only: 'SUM 3 4'.

Logo stores procedure (operation) names and names of values distinctly. This allows constructions like: 'PRINT :VALUE:', which does not execute the 'VALUE' operation. Procedure text is only accessible via certain operations, but, with these, programs can modify themselves. A schematic of Logo's memory space appears in Figure 3.

A procedure is defined by the user with the 'TO' operation, which expands Logo's internal dictionary of the operations it can obey. A procedure definition takes the form of a title and a body. The title states the name of the new operation and the names to associate with any values it should expect as arguments. The body consists of a sequence of numbered lines. Each line is itself a Logo command; line numbers serve as editing handles and define the sequence in which the lines will be executed.

Evaluation of a Logo command implies execution of at least the operation(s) named and perhaps other commands, as arguments or possible side effects. Because of the prefix nature of Logo's syntax, Logo processes a command in two steps, first parsing left to right until a subcommand is found which has sufficient input arguments for execution, and then returning values from right to left as it executes any subcommands suspended for want of computed arguments. Often these two steps will alternate as a command is obeyed. Implicit in this processing scheme is a mixture of evaluation and execution mediated by an ability to preserve, and later restore, the information associated with any subcommand(s) whose execution must be suspended when other execution is called for. This processing naturally extends to user procedures which call other procedures, use primitives, or call themselves. A clarifying example follows.

The underlying structure which allows Logo's form of processing is the "execution stack" in Figure 3 -- a standard "last-in-first-out" data structure (e.g. Evey, 1963). Information enters and leaves the stack only via its "topmost" cell. As Logo preserves information pertaining to commands whose execution has been suspended, the stack acquires the execution history of a program as a list of things left undone -- most recent history on top. Normally, information added to the stack is later removed, because suspended commands are eventually obeyed. The stack is usually empty both before and after a user's command to Logo has been obeyed.


user procedure definitions (editable)
UUUUU
user abbreviations (modifiable)
UUUUU
primitive Logo operations
execution stack
Logo names & abbreviations
Logo string space
user names (modifiable)
UL
L
uL
U
u

Legend

LLL denotes a link that Logo can exploit
UUU denotes a link that the user can exploit
uuu denotes a link to which the user has partial access

Fig. 3. Schematic of Logo's Memory Space.

Readers unfamiliar with this evaluation method are urged to use Figure 4 to follow the effect of the command: 'PLAY', assuming the two procedure definitions:

```
TO PLAY
10 TYPE "WHAT NUMBER AM I THINKING OF?"
20 PRINT GUESS RANDOM REQUEST
END

TO GUESS :IT: :THAT:
10 IF EQUALP :THAT: :IT: THEN RETURN "WOW" ELSE PRINT "TRY AGAIN"
20 RETURN GUESS :IT: REQUEST
END
```

They are printed here exactly as a user would have typed them to Logo. As part of our results, we will consider student errors related to Logo's command-evaluation method.

Whenever 'GUESS' is executed, it expects to receive two input values (via the stack) which it will associate with (bind to) the names 'IT' and 'THAT'. Whenever 'IT' or 'THAT' is evaluated (e.g. line 10), Logo searches the stack for the <u>first</u> (latest) such binding.* The 'IF ... THEN ... ELSE ...' structure is an execution selector -- 'IF' receives either "TRUE" or "FALSE" from 'EQUALP', causing Logo to execute either the command marked by 'THEN' or that marked by 'ELSE'. Here 'GUESS' either may return a value ("WOW") by replacing its own name in the stack, or it may defer returning a message and call on itself (line 20) recursively. This creates a new copy of 'GUESS' on the stack without destroying the old copy. When 'RETURN "WOW"' is executed by

---

* Logo is derived from Lisp, so inputs are not "local variables" in the Algol sense, although locals may be defined in Logo.

(a) Logo about to execute line 10 in PLAY

```
top---⊥_"WHAT NUMBER AM I THINKING OF?"
      ⊥_TYPE (awaiting 1 input)
      ⊥_PLAY (resume after line 10)
```

(b) line executed

```
⊥_PLAY (resuming)
```

(c) starting RANDOM

```
⊥_RANDOM
⊥_GUESS (awaiting 2 inputs)
⊥_PRINT (awaiting 1 input)
⊥_PLAY (resume after line 20)
```

(d) starting GUESS

```
⊥_2 (returned by REQUEST)
⊥_3 (returned by RANDOM)
⊥_GUESS
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(e) starting EQUALP

```
⊥_3 (latest value for "IT" on stack)
⊥_2 (latest value for "THAT")
⊥_EQUALP
⊥_IF (awaiting 1 input)
⊥_:THAT: = 2 (GUESS' 2nd input binding)
⊥_:IT: = 3 (GUESS' 1st input binding)
⊥_GUESS (resume after line 10)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(f) starting IF

```
⊥_"FALSE"
⊥_IF
⊥_:THAT: = 2
⊥_:IT: = 3
⊥_GUESS (to resume)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(g) starting the "ELSE" part

```
⊥_"TRY AGAIN"
⊥_PRINT
⊥_:THAT: = 2
⊥_:IT: = 3
⊥_GUESS (to resume)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(h) GUESS calling GUESS

```
⊥_3 (returned by REQUEST)
⊥_3 (latest binding of "IT")
⊥_GUESS (new copy)
⊥_RETURN (awaiting 1 input)
⊥_:THAT: = 2
⊥_:IT: = 3
⊥_GUESS (to resume)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(i) copy returning "WOW"

```
⊥_"WOW"
⊥_RETURN
⊥_:THAT: = 3
⊥_:IT: = 3
⊥_GUESS (resume after line 10)
⊥_RETURN (waiting)
⊥_:THAT: = 2
⊥_:IT: = 3
⊥_GUESS (to resume)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(j) GUESS returning "WOW"

```
⊥_"WOW"
⊥_RETURN
⊥_:THAT: = 2
⊥_:IT: = 3
⊥_GUESS (to resume)
⊥_PRINT (waiting)
⊥_PLAY (to resume)
```

(k) about to complete PLAY

```
⊥_"WOW"
⊥_PRINT
⊥_PLAY (to resume)
```

Fig. 4. An Example of Logo's use of Its Execution Stack.

some copy of 'GUESS', a virtual bucket-brigade sends "WOW" back to 'PRINT' (in 'PLAY') by removing information from the stack as each suspended 'GUESS' returns (line 20). A procedure may also simply execute without returning a value. 'PLAY' does this by default (by running out of lines); equivalently, it could have contained a line 30 which just said: 'DONE'. Alternatively line 30 could say: 'PLAY'. The reader should pursue the effect of that change.

2.1.2 Editing and Debugging Facilities

In both Simper and Logo, editing may be categorized as either: (1) line editing, or (2) program editing. Most of the basic line-editing abilities in either language arise from a machine instruction peculiar to the IMSSS time-sharing system.

A "line" is any string of keyboard characters terminated by any of a small set of keys (e.g. carriage return). As Table V suggests, a line may be edited or extended before such termination by any of several "control-characters". As the Table indicates, some commands were implemented only in Logo. Commands like "control-N" mesh well with Logo's sentences, but were not needed in Simper because of the short, simple nature of Simper's instructions.

Program editing in Simper is mediated by: (1) program-displaying, and (2) program-altering commands (again see Table IV). 'DUMP', 'LIST' and 'NAMES' fall into category (1). Since a program is stored as numerals in memory (viewable with 'DUMP'), 'LIST' must translate numerals back into assembly language whenever they appear to be legal

Table V

Simper/Logo Line-editing Commands (* means Logo only)

Name	Action
control-A or rubout	erases the previous character typed
control-W	erases the previous word typed
control-X	erases the whole line (also control-U in Simper)
control-R	retypes the present line minus deletions
linefeed	continues a line beyond 72 characters
return or altmode	terminates a line (altmode is also known as "escape" or "enter")
control-N*	insert (into the present line) the next word from the previous (or edited) line
control-S*	skip the next word from the previous (or edited) line
control-E*	insert everything remaining in the previous (or edited) line into the present line

instructions. This is true even for instructions which can be generated ambiguously. For example: 'ADD A 100', 'ADD A IT+2' (where 098 is named "IT") and "101 :ADD A .-1" all translate into: '2110100'. How then to translate '2110100' back into the form that the user obviously preferred? That is facilitated by a table, parallel to memory. Among other things, each cell in this table holds information about the nature of the address referenced by the instruction in its companion memory cell (see Figure 5). If the user types an assembly language instruction containing a relative and/or symbolic address, the appropriate entries are made in the table as the instruction is assembled into memory. Note from Figure 5 that if an address field

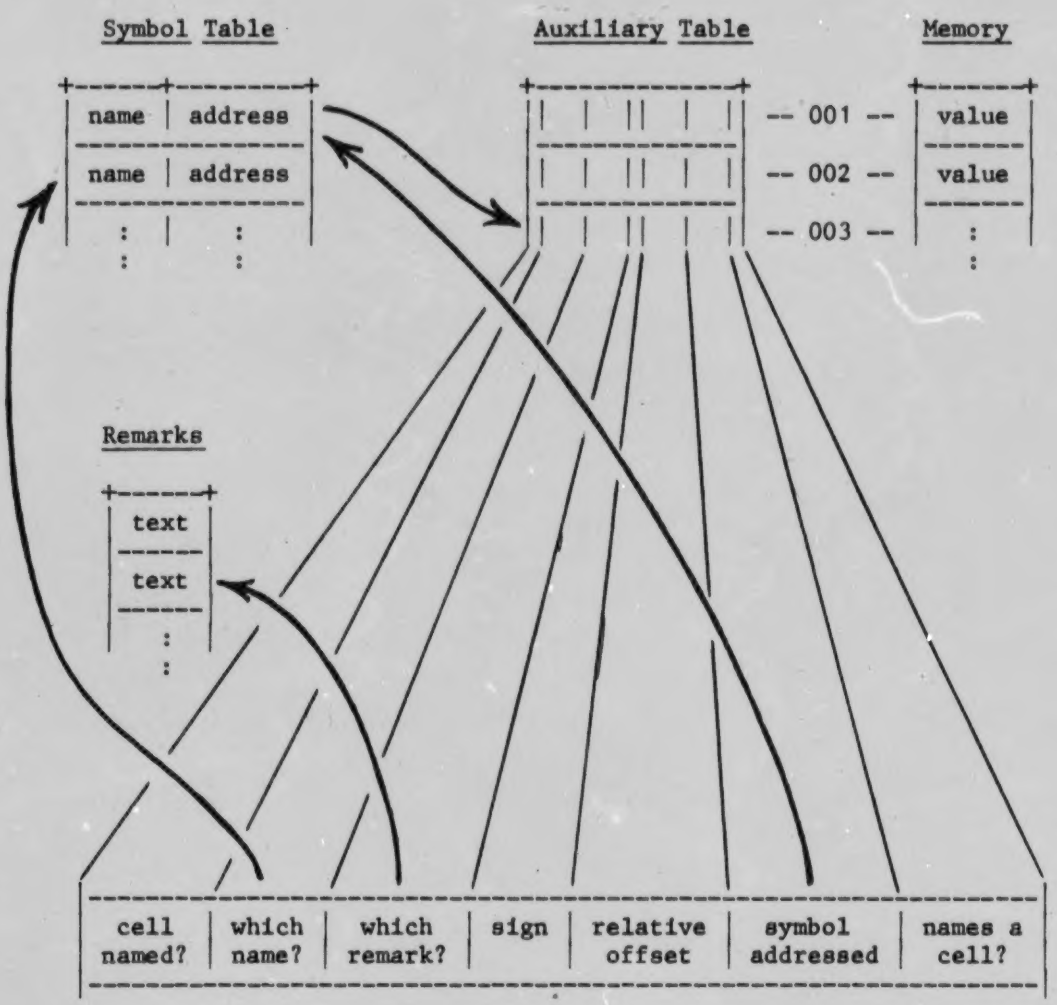


Fig. 5. Supporting Structure for the Simper Assembler.

contains a symbol, a pointer into the symbol table is stored; if it contains a relative offset, the amount and sign are stored. Furthermore, if a cell has a name or a remark (comment) associated with it, that information is stored as well. Although the auxiliary table is inaccessible to the user (normally ignorant of 'DEBUG'), it provides a valuable service to 'LIST' for its task of reconstructing assembly listings. It also facilitates other editing services, as we will mention. The symbol table may be inspected by the user with the 'NAMES' command -- the addresses and contents bound to all symbols are displayable.

A program may be altered by any of: 'EDIT', 'SLIDE', 'SCRATCH', 'NAME' and 'FORGET'. 'EDIT' effectively places the user anywhere in Simper's memory so that any number of contiguous locations can be given new contents. When all the requested locations have been visited, 'EDIT' returns the user to the address from which the original command was given. For example: "003 :EDIT 5:7" would prompt "005 :" through "007 :" and then return to "003 :". One may see the present content of a location to be edited by terminating an 'EDIT' command with the "altmode" key rather than the "carriage return". The standing rule in Simper is that terminating a command with "altmode" is equivalent to requesting whatever extra information that command can supply. The extras are parenthesized in Table IV.

'SLIDE' is the most powerful editing command available in Simper. It allows one to relocate a block of code in memory without having to do additional editing to fix up addresses which point into or out of that block. A program is essentially executable after any 'SLIDE'. Examples appear in Figure 6. The command can move a block (a contiguous group of non-zero memory values) either forward (to higher addresses) or backward (to lower addresses) in memory. A move forward allows the user to create an empty region for insertion of new code. A move backward destroys any old code in the region covered by the new

(a) Opening up space for new instructions:

<u>before</u> the command: 'SLIDE 3:5', and <u>after</u>

```
001 :ASK A                             001 :ASK A
002 :MUL A 10                          002 :MUL A 10
003 :ADD A 6                           003 :
004 :WRI A                             004 :
005 :PUT P 1                           005 :ADD A 8
006 :9                                 006 :WRI A
007 :                                  007 :PUT P 1
008 :                                  008 :9
009 :                                  009 :
010 :2                                 010 :2
```

(b) Replacing an undesired sequence of code:

<u>before</u> the command: 'SLIDE 5:3', and <u>after</u>

```
001 :ASK A                             001 :ASK A
002 :MUL A 10                          002 :MUL A 10
003 :JUM A .+1                         003 :ADD A 6
004 :PUT A 1                           004 :WRI A
005 :ADD A 8                           005 :PUT P 1
006 :WRI A                             006 :9
007 :PUT P 1                           007 :
008 :9                                 008 :
009 :                                  009 :
010 :2                                 010 :2
```

Fig. 6. Two Uses of Simper's 'SLIDE' Command.

position of the relocated block. If one wishes to erase but not move a certain region in memory, then 'SCRATCH' is useful. Both 'SLIDE' and 'SCRATCH' appropriately revise the supporting tables (in Figure 5).

Symbols are created with 'NAME' and destroyed with 'FORGET'. Any memory cells may be named. 'NAME' writes the symbolic name and it's associated cell's address into the symbol table. A symbol also may spring into existence if it is used in the address field of an instruction. In such a case, the name is entered into (or matched in) the symbol table and a pointer to it is installed in the proper cell (Figure 5, subcell: "symbol addressed") of the auxiliary table. If 'NAME' has not yet been used to tie that symbol to some memory cell, then both the address field of the assembled instruction and the "address" subcell of the symbol table entry must remain blank. This condition is indicated by setting the "names a cell?" subcell to "no". Should the symbol ever be tied to a cell, the assembler searches memory and "fixes up" the address fields of instructions so marked. The association of a symbol may be moved from one memory cell to another with 'NAME'. This may also result in reassembly of the address fields of some instructions.

'FORGET' cannot erase a name from the symbol table as long as that name is referenced in any address field. This is designed to protect the user, lest he or she suddenly be confronted with many instructions, in a complicated program, whose address fields are redundant and/or meaningless.

Some miscellaneous commands are available to the Simper programmer. A program may be saved on and later retrieved from the operating system's long-term storage by using 'SAVE' and 'GET' respectively. This entails saving only the essentials needed to reconstruct the tables depicted in Figure 5. Another command: 'FIELDS' can be used to reduce the typing needed for instructions which use the A register. This command is a toggle which, when turned on, tells the assembler: "I want

'A' in the register field unless I say otherwise." For example, with the toggle on, the first program in Figure 6 could have been typed:

```
ASK
MUL 10
ADD 6
WRI
PUT P 1
```

and it would have been so listed (by 'LIST') as long as 'FIELDS' was not used to reset the toggle. This feature provides a convenient simulation of a single-register machine. Finally, one of the most important commands available to the user is "control-G", which aborts or nullifies the effect of any other command and returns the user to the outer level of the Simper interpreter.

Since the basic piece of any Logo program is the procedure, program editing in Logo amounts to procedure creation, deletion and modification. 'TO' and 'ERASE' handle the first two activities, while 'EDIT' and several contingent operations (Table VI) handle the latter. The interpreter enters editing mode (signified by the prompt: "@", see Figure 1) whenever a 'TO' or an 'EDIT' command is given. During editing, any other executable Logo command may be given. In fact, some operations (indented in Table VI) only have meaning in this context, and several expect input messages that define their scope (Figure 1 should be examined together with Table VI). However, the syntax of these commands does not match that outlined in the previous section. In the available version of Logo, inputs to operations like 'TO' or 'EDIT' are <u>not</u> subject to normal evaluation rules; rather, they are quoted by default. For instance, it is not possible to directly pass the name of

Table VI

Logo's Procedure Editing and Debugging Commands

Name	Action
TO	begin defining a new procedure
EDIT	begin modifying an existing procedure
TITLE	redefine the name of the procedure and its inputs
EDIT TITLE	change part of the title
LIST TITLE	display the title
EDIT LINE	change part of any line in the procedure
ERASE LINE	delete any line
LIST LINE	display any line
END	stop editing the procedure's definition
LIST	display any procedure's definition
ERASE	delete any procedure's definition or trace
ERASE ALL PROCEDURES	delete all definitions
LIST ALL PROCEDURES	display all definitions
LIST CONTENTS	display the titles of all defined procedures
LIST ALL ABBREVIATIONS	display the user's abbreviations for all operations
TRACE	display a procedure's arguments/returned value whenever it is executed
BREAK	halt execution (same as control-G)
EXIT	halt and print a message
GO	continue execution

Indented commands may only be given after editing has been begun with 'TO' or 'EDIT'.

a procedure to be edited (e.g. 'Y') via a value bound to some variable (e.g. 'EDIT :X:' isn't legal, 'EDIT Y' or 'DO SENTENCE "EDIT" :X:' are).

Logo saves the full text (expanding abbreviations) of lines and titles of procedures as the user defines them. The interpreter does not regenerate listings from some internal code as Simper necessarily must. This conveys two benefits: (1) the user may write procedures which in turn copy, edit or write new procedures, and (2) the Logo interpreter readily brings forth any parts of lines to be edited. (1) derives from normal Logo primitives, while (2) derives from the implementation of "control-N", "control-S" and "control-E" (Table V). These three commands can access a stored line and control its injection into the user's typing. Since procedure lines and titles are stored, old lines can be used to construct new ones. Suppose for example, that one wishes to edit the one line in the existing procedure listed below and add a new, similar line.

```
TO WELCOME
10 SAY "HELLO THERE"
```

The command: 'EDIT LINE 10' causes the line number "10" to be printed <u>and</u> inserted into Logo's input buffer just as if the user had typed it. Thus the line number may be erased or changed. At this time, Logo has grabbed the existing text of line 10 and knows 'SAY' to be its first word. Here is the editing sequence which produces line 20 by using line 10 ("↑" means "control-", Logo's typing is in lower case, deleted characters are in brackets):

```
10 [ 01]20 ↑Nsay "↑S↑Nthere" [ "] GOES A WELCOME"
```

Normally, the control characters are not printed on the user's terminal and, on graphics displays, deleted characters simply disappear. The procedure would now have the lines:

```
TO WELCOME
10 SAY "HELLO THERE"
20 SAY "THERE GOES A WELCOME"
```

If the user were now to type "control-E", the entire text of line 20 would be made available again for editing. This is because Logo always sequesters a copy of the last line terminated by the user. Its text is available at any time until another line terminator is typed. Because all line-editing commands operate independently of procedure-editing commands, one can, for instance, type:

```
SAY "GOODBYE"
30 ↑Esay "goodbye"
```

to test a command before storing it as a line in the procedure being edited. Such access to previously typed lines can save the user much typing and reduce typing errors.

Logo also has provision for saving programs on and restoring them from the operating system's file storage (see Table VII). The structure is more complex than Simper's, allowing all of Logo's memory (all procedures, bindings and abbreviations) to be saved as an "entry" on a file. Each file may have many entries and more than one entry may be read into memory at once, thus allowing programs to be combined. Files may be examined and entries may be erased without being read into active memory. The syntax of these commands is like that of the

Table VII

Logo's File-manipulation Commands

Name	Action
SAVE	replace an entry on a file with the current contents of memory
GET	append the content of an entry to memory
LIST FILE	display the entry names in a file
LIST ENTRY	display everything in an entry
LIST PROCEDURES	display only the procedures in an entry
LIST CONTENTS	display the titles of an entry's procedures
LIST ABBREVIATIONS	display the abbreviations in an entry
ERASE ENTRY	delete an entry from a file
COPY	copy a text file to or from a file entry

editing commands discussed above. Later, we will discuss the effect on students of that and of the relative complexity of Logo's filing system.

The user can supply new abbreviations, or use those which Logo has built in, for relatively wordy Logo operations such as those listed in Tables III, VI and VII.

Debugging. Program debugging in Simper is facilitated primarily by using the register displaying option of the 'RUN' command (Table IV), which is activated by terminating the command with the "altmode" key. Also, the user may stop a program at any point with "control-G", examine memory and register values with 'DUMP', 'LIST' or 'NAMES', perhaps do some editing and then continue the execution with 'GO'. Stopping a program in this way does no violence to the state of the machine; the program counter (P) is always saved to anticipate the use of 'GO'. The user may continue execution for a specific number of instruction cycles (e.g. 'GO 5') and/or alternate execution periods with the register display on and off. He or she also may run selected portions of a program (e.g. 'RUN 4:12') to check their operation. In Figure 7, we show a typical display for a run of the first program in Figure 6.

```
007 :RUN$ ("$" denotes altmode)

13:04:12 (the time)

EXECUTING 1 TO 500

 P:    A:    B:   INSTR:
------------------------
 1      0     0   ASK A         INPUT NUMBER:4 ("4" typed by user)
 2      4     0   MUL A 10
 3      8     0   ADD A 6
 4     17     0   WRI A         NUMBER=17
 5     17     0   PUT P 1
 1     17     0   ASK A         INPUT NUMBER:0
 2      0     0   MUL A 10
 3      0     0   ADD A 6
 4      9     0   WRI A         NUMBER=9
 5      9     0   PUT P 1
 1      9     0   ASK A         INPUT NUMBER:-4
 2     -4     0   MUL A 10
 3     -8     0   ADD A 6
 4      1     0   WRI A         NUMBER=1
 5      1     0   PUT P 1
 1      1     0   ASK A         INPUT NUMBER:↑G (user aborts)

...15 INSTRS IN 1.100 SEC

007 :GO 4 (continue a bit with no display)
2
13

...4 INSTRS IN .042 SEC
```

Fig. 7. Displaying a Simper Program's Activity.

Without this display, the user's program has full control over the formatting of its output.

Program debugging in Logo centers on the use of the commands 'TRACE' through 'GO' of Table VI. "Control-G" and 'GO' have the same functions in Logo as in Simper, although 'GO' did not always work successfully in our version of Logo. 'EXIT' and 'BREAK' are simply ways of returning control to the user when some condition defined by the user occurs. 'TRACE' is the most important debugging command in Logo. It allows the user to follow a particular procedure's (but not a Logo primitive's) execution history, observing its arguments when it is called and the value it returns when it is done. For recursive procedures, each copy is so observable. Figure 8 shows an example generated by the commands: 'TRACE ACKERMAN' and 'PRINT ACKERMAN "XX" "Y"', that executed the procedure:

```
TO ACKERMAN :X: :Y:
10 IF EMPTYP :X: THEN RETURN WORD :X: "Y"
20 IF EMPTYP :Y: THEN RETURN ACKERMAN BUTFIRST :X: "Y"
30 RETURN ACKERMAN BUTFIRST :X: ACKERMAN :X: BUTFIRST :Y:
```

realizing a string example of Ackerman's function. In the figure, inferior context (a copy) is indicated by indentation. The reader should try to justify the traced execution sequence with the procedure's definition and its first call. Notice that 'ACKERMAN "XX" "Y"' first causes the execution of 'ACKERMAN :X: BUTFIRST :Y:', at the end of line 30; that call is the next indented line in the trace.

```
_TRACE ACKERMAN
_PRINT ACKERMAN "XX" "Y"
ACKERMAN OF "XX" AND "Y"
 ACKERMAN OF "XX" AND ""
  ACKERMAN OF "X" AND "Y"
   ACKERMAN OF "X" AND ""
    ACKERMAN OF "" AND "Y"
    ACKERMAN RETURNS "YY"
   ACKERMAN RETURNS "YY"
   ACKERMAN OF "" AND "YY"
   ACKERMAN RETURNS "YYY"
  ACKERMAN RETURNS "YYY"
 ACKERMAN RETURNS "YYY"
 ACKERMAN OF "X" AND "YYY"
  ACKERMAN OF "X" AND "YY"
   ACKERMAN OF "X" AND "Y"
    ACKERMAN OF "X" AND ""
     ACKERMAN OF "" AND "Y"
     ACKERMAN RETURNS "YY"
    ACKERMAN RETURNS "YY"
    ACKERMAN OF "" AND "YY"
    ACKERMAN RETURNS "YYY"
   ACKERMAN RETURNS "YYY"
   ACKERMAN OF "" AND "YYY"
   ACKERMAN RETURNS "YYYY"
  ACKERMAN RETURNS "YYYY"
  ACKERMAN OF "" AND "YYYY"
  ACKERMAN RETURNS "YYYYY"
 ACKERMAN RETURNS "YYYYY"
ACKERMAN RETURNS "YYYYY" (to PRINT)
YYYYY
_
```

Fig. 8. Tracing a Logo Procedure's Activity.

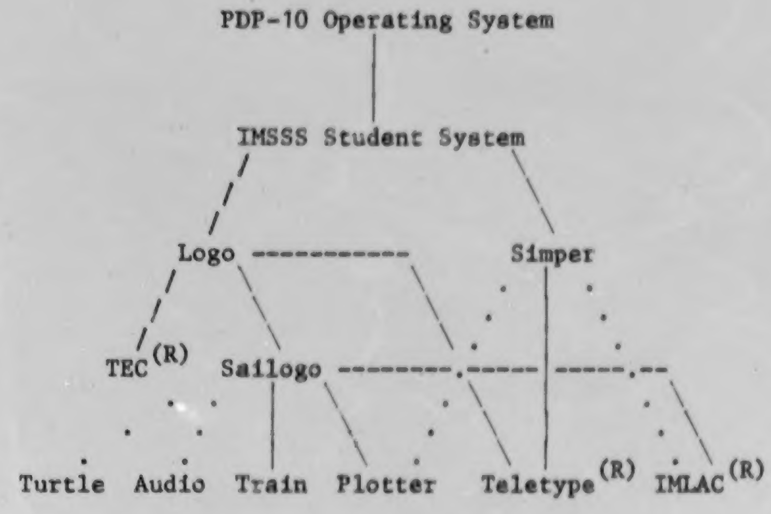
Ackerman's function was chosen because it provides a general exercise for Logo's tracing ability. The Logo procedure in Figure 1 and that used for Figure 4 are not as suitable because they are "last-line" recursions. The processes they realize are essentially iterative. They do not make use of the local context which is saved when a new copy of a procedure is recursively called and the calling copy is suspended. In contrast, note the sequences like: 'ACKERMAN

RETURNS ...', 'ACKERMAN OF ...' in Figure 8 (generated by the recursive calls in line 30) and note the values supplied as arguments for each call. In fact, some operations, like Ackerman's function, cannot be expressed clearly in iterative fashion and thus are difficult to formulate in the syntax of languages (e.g. Basic or Fortran) which do not provide for recursively defined algorithms.

The user can, of course, trace as many procedures as necessary for debugging a program. Moreover, debugging during program synthesis is facilitated by the line-editing commands like "control-N" that were discussed earlier. Since Logo will always execute a direct command, even when defining a procedure, the user can try a number of command lines until one has the desired effect, and then he or she can type a line number and "control-E" to copy that last, workable line into the procedure's definition. This is helpful when writing procedures which draw or engage in some actions that must be subjected to fine tailoring.

2.2 Peripheral Devices

In the following sections we provide information about the various terminals and controllable devices available to Logo and Simper students both during and after the experiment (Figure 9). We also mention examples of how each device can be employed in solutions to posed programming problems. On occasion, sample programs are included -- the reader should refer back to the previous sections if their meaning is unclear. In a later section we will evaluate the usefulness of each device based upon our experimental observations.



Dotted lines mark connections made after the 1973 summer experiment.

Fig. 9. Programming System Structure.

For communicating with devices like the IMLAC(R), Train and Audio, the machine-language Logo interpreter was modified to dispatch pertinent commands to another program, Sailogo (Figure 9). This program runs as a coroutine (an "inferior fork" in Tenex terminology) to Logo. Logo and Sailogo each possess 256-kiloword, virtual memory spaces which are independent except for one shared "page" of 512 words. This shared space is used for the intercommunication of commands, results and error messages. When Logo traps a command to be interpreted by Sailogo (e.g. 'SAY "HELLO"'), it puts the operation code and any arguments into the shared page, starts the Sailogo process, and suspends itself until Sailogo replies and terminates with an appropriate response. Hence, Logo's primitive control of special devices is realized by Sail procedures.

2.2.1 Standard Alphanumeric Terminals

The slow (10-characters-per-second), noisy, reliable and inexpensive Model 33 Teletype$^{(R)}$ was the basic means for communicating with Logo and Simper for students in three of the experimental groups. In spite of its obvious drawbacks it was in plentiful supply and provided paper printout for projects whose results students wanted to take home (e.g. posters). After they had mastered the basic concepts of Logo and Simper, students could spend some time using the more exotic terminals and devices as they were available. Some students retained a particular liking for teletypes, because the mechanical bedlam generated by an operating teletype fascinated them.

The TEC$^{(R)}$ is a fast (several hundred characters-per-second), text-oriented, video display with capabilities for cursor positioning and line or character insertion and deletion. Because these terminals were neither usually available nor centrally located, students used them infrequently. However, we will outline several examples of Logo programs which exploit the TEC's capabilities (see Table VIII, or Appendix 1.1 for a summary of the relevant IMSSS Logo commands). For example, one student defined a "TEC-turtle" which interpreted commands similar to the robot and IMLAC$^{(R)}$ line-drawing instructions. His turtle was simply an outline in characters. It could rotate in multiples of 45 degrees and leave a trail of characters as it moved. Another student developed a "twinkling sky" program which randomly placed and erased characters on a TEC's screen.

Table VIII

IMSSS Logo/TEC$^{(R)}$ Commands

Name	Action
CLEAR	erase screen and put cursor in upper left corner
UP	move cursor up one line (with wraparound)
DOWN	move cursor down one line (with wraparound)
LEFT	move cursor left one character (with spiral wraparound)
RIGHT	move cursor right one character (with spiral wraparound)
HOME	put cursor at upper left corner of screen
MOVEXY	put cursor at an absolute screen position (in 80 by 24)
BLINKON	start screen blinking to right of and below cursor
BLINKOFF	terminate blinking region at present cursor position
BOX	type a "box" character (all character dots on)
DELETECHAR	erase character on cursor and move rest of line left
DELETELINE	erase line cursor is on and move lower lines up
ERASEDOWN	erase screen to right of and below cursor
ERASERIGHT	erase rest of line to right of cursor
INSERTCHAR	insert a blank character at cursor position
INSERTLINE	insert a blank line at cursor position

The activity of procedures which manipulate Logo words and sentences can be visually seen by writing the procedures so that they provide graphic output of internal string machinations. For example, students could watch a program move the TEC's cursor from character to character as it searched for certain characters which it then deleted both from the Logo word and the display screen. Elevator (see Appendix

1.1) and "drunken walk" simulations were also done. With the TEC display, it is possible to produce simple animations of a program's activity.

2.2.2 Vector Graphics Terminals

From 9:00 AM to 11:00 AM on weekdays, five IMLAC$^{(R)}$ PDS-1 vector displays were reserved for two of the five experimental groups. The PDS-1 is a computer in its own right, but students used it only as a "smart" interface to the PDP-10 (see Appendix 1.2 for details on the PDS-1 and on the Logo interface to our graphics system). The IMLAC-graphics line-drawing system was implemented to emulate many abilities of the robot turtle developed at MIT and BBN (Feurzeig and Lukas, 1972b). It lacks the mechanical robot's touch sensors (although these could be simulated by graphic constraints), but allows movement to be specified by "X,Y" end-points in addition to the turtle's normal, roving-polar-coordinates scheme (in which movement is specified by 'FRONT' and 'BACK' along an angular heading changed by 'RIGHT' and 'LEFT'). For example, to draw a square, one might write the following Logo procedure:

```
TO SQUARE :SIZE:
10 FRONT :SIZE:
20 RIGHT 90
30 FRONT :SIZE:
40 RIGHT 90
50 FRONT :SIZE:
60 RIGHT 90
70 FRONT :SIZE:
80 RIGHT 90
END
```

Students can change the IMLAC turtle's appearance with 'SEE', 'HIDE', 'POKE' and 'UNPOKE' (see Table IX and Appendix 1.2 for a description of IMSSS Logo's turtle-graphics primitives). The graphics turtle can leave a trace ('PENDOWN'), or not ('PENUP'), as it moves. The last lines drawn may be erased by 'ZAP' and 'ZIP' commands, permitting limited picture editing as well as primitive animation. For instance, one student produced a short sequence showing a fuse "burning" down (disappearing into) and exploding a firecracker.

'PLOT' allows one to direct the effects of most graphics commands to either an HP7202A plotter or a robot turtle. Most students highly valued the ability to reproduce on paper what their programs had drawn on the display screens.

During the experiment, it became apparent that more powerful animation abilities would be possible and might serve as strong motivation for more complex student projects. At the end of the summer experiment, we added genuine animation to Logo (see Table X or Appendix 1.2 for a description of Logo's animation primitives). By typing 'SNAP', the student instructs the graphics system to save the effects of all subsequent graphics commands as a display subroutine within the IMLAC. 'ENDSNAP' terminates the subroutine. The result is a numbered "snapshot" which can be shown anywhere on the IMLAC screen. 'SHOWSNAP' shows such a snapshot at the turtle's current screen location. For the purposes of erasing (with 'ZAP' or 'ZIP'), a snapshot is equivalent to a line segment. Rotation and scaling can be achieved by making several snapshots of the same object in different orientations or sizes and then showing them successively in a "movie". Thus, a simple recursive

Table IX

IMSSS Logo Turtle-Graphics Commands

Name	Action
CLEAR	erase the text area of the screen
WIPE	erase any drawing and put turtle home
SEE (HIDE)	make the turtle appear (disappear)
PENDOWN (PENUP)	enable turtle to draw visible (invisible) lines
PENP	return "TRUE" if turtle's pen is down, "FALSE" otherwise
POKE (UNPOKE)	stick out (pull in) turtle's head
HOME	move turtle to home position defined by SETTURTLE
FRONT (BACK)	move turtle forward (backward) a specific distance
LEFT (RIGHT)	rotate turtle left (right) specific number of degrees
SETHEADING	point turtle on a specific angular heading
ASETX (ASETY)	move turtle horizontally (vertically) to an absolute screen position
ASETXY	move turtle horizontally and vertically to a position
RSETX (RSETY)	move turtle horizontally (vertically) a relative amount
RSETXY	move turtle relative to its present screen position
THERE	equivalent to an ASETXY and a SETHEADING
HERE	return turtle's current position and angular heading
ARC	make turtle draw an arc of specified radius and sense
ZAP (ZIP)	erase last turtle move(s) up to a visible line segment
PLOT (UNPLOT)	(do not) direct turtle commands to robot or plotter
SETSCALE	set screen resolution in units-per-inch
SETTURTLE	set both scale and home position on screen
WRAP	set up screen boundaries for wraparound
COMPRESS	shorten IMLAC display list (precludes use of ZAP or ZIP)

Table X

IMSSS Logo Animation Commands

Name	Action
SNAP	wipe screen and begin creating a numbered "snapshot" of whatever drawing (less erasures) is subsequently done
ENDSNAP	finish defining current snapshot and wipe screen
ERASESNAP	delete specified snap and its number
WHATSNAPS	return a sentence of currently used snapshot numbers
SHOWSNAP	display specified snapshot at turtle's screen position
PUTSNAP	identify a snapshot with an old or new "object" at a specific screen position, or move or erase an object
MOVESNAP	move an object (with wraparound) a relative distance on a relative heading and return object's final, absolute position
WIPESNAPS	wipe screen and erase all snapshots and objects

procedure for moving an object, referenced by a snapshot number, across the screen might be:

```
TO WALK :SNAPNUMBER:
10 SHOWSNAP :SNAPNUMBER:
20 ZAP
30 FRONT 10
40 WALK :SNAPNUMBER:
END
```

(The reader might try to imagine the scene produced if line 20 were omitted).

With respect to additions and deletions, the 'SNAP'/'SHOWSNAP' scheme establishes a stack (last-in-first-out) ordering on the elements of a scene. It proved adequate for many simple animation projects, but it prevents placing a snapshot independently of the turtle and it precludes erasing a particular snaphot without first erasing and then restoring all picture elements that were drawn after that snapshot. Given the nature of the IMSSS Tenex time-sharing system, this scheme usually requires too much time to communicate, from Logo to IMLAC, all the data needed for complex yet brisk animations (see also Section 5).

In order to manipulate snapshot occurrences independently of their displayed order, and to reduce intercommunication needs, we provided two additional operations. 'PUTSNAP' creates an "object" which is a snapshot placed at a particular "X,Y" location on the screen. For example: 'PUTSNAP "5 1" "0 0"' associates snapshot 5 with object 1 at screen location 0,0. The *relative* location of this object now can be changed with 'MOVESNAP'. The snapshot associated with an object simply defines that object's *current* appearance. Thus, subsequent 'PUTSNAP' commands can change either the appearance or the absolute location of an object already on the screen.

A large saving in communication time resulted from this design; complex animations became viable even in a normal time-sharing environment. The 'WALK' program shown above could be rewritten as:

```
TO WALK :OBJECTNUMBER:
10 MOVESNAP :OBJECTNUMBER: "10 0"
20 WALK :OBJECTNUMBER:
END
```

Although the animation facilities were not used by students during the summer experiment reported upon here, they were used in a later experiment (Cannara, 1975). A number of students from the summer experiment continued to work with Logo after school began, influencing some aspects of the developing animation system. A two-minute, black-and-white film about Logo/IMLAC graphics and animation is available from IMSSS.* Figure 10 is taken from that film.

Students used Logo animation to produce such things as a flyable helicopter, a rocket launch, animated tic-tac-toe and a movie of throbbing polygons. An example program appears in Appendix 1.2.

A Windmill Simulated with Logo/IMLAC(R) Animation

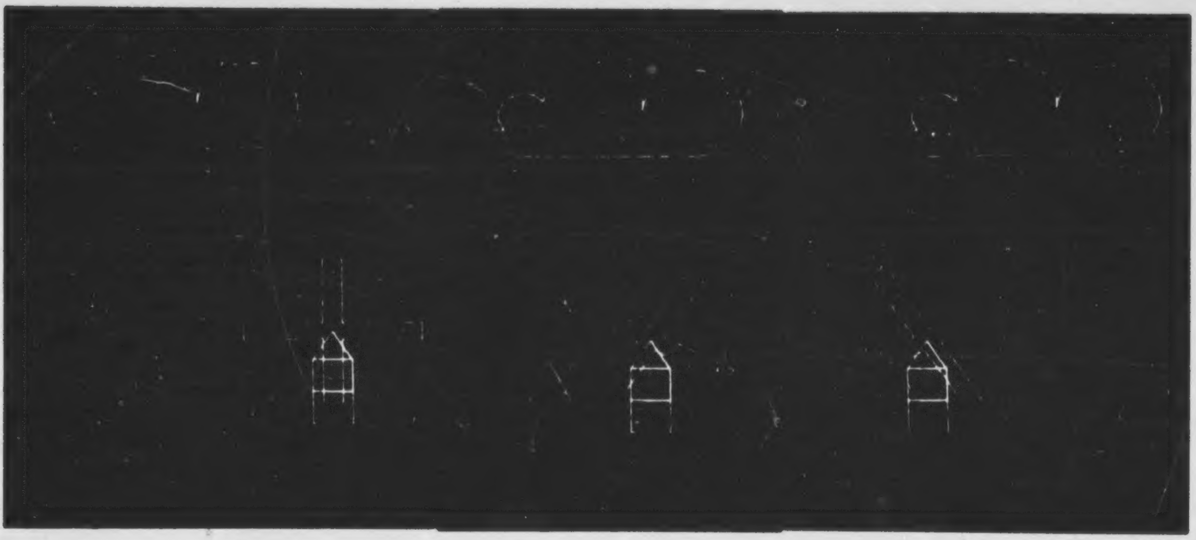


Fig. 10. Successive Frames from a Logo-Animation "Movie".

---

* We are indebted to Pat Crawley of the Stanford Communications Department for producing this film, and to Adam Grosser, Greg Hinchliffe and Steve Spurlock for their imaginative programming.

2.2.3 Output Devices: Plotter, Turtle, Train and Audio

Twice, near the middle and end of the experiment, a Hewlett-Packard model 7202A plotter was available to our students. With this device, students could produce permanent pictures on paper. This particular plotter has a resolution of one part in ten-thousand and directly accepts alphanumeric (ASCII) strings for line and point plotting, making it an extraordinarily easy device to connect to existing systems. By first typing 'PLOT' and an appropriate teletype number to Logo, a student can execute almost any Logo-graphics commands with ink on paper. Erasing and animation commands (e.g. 'ZAP') have no effect. Students can use any type of terminal and still have their drawings appear on the plotter. This can be used to encourage students to write and debug storable procedures rather than to just draw by direct Logo commands.

A robot turtle, music box and their interface (all manufactured by General Turtle Inc. of Cambridge, Mass.) also were available to our students for several days at the end of the summer experiment, and again, during winter and spring -- 1973-1974. Like the plotter, the robot/music-box interface is straightforward to use because it interprets ASCII characters as commands. These devices were not used as much in the experiment reported upon here as in the subsequent one, so further details on them are confined to Appendix 2.1.

During winter and spring, 1970-1971, a Marklin$^{(R)}$, HO-gauge, electric-train layout was constructed at IMSSS (see also Goldberg, Levine and Weyer, 1974). It uses a special interface which decodes characters, sent by PDP-10 programs, as commands for setting switches and controlling the train's motion (see Appendix 2.2). The interface also responds to queries about the presence of cars within a number of distinct regions of track (blocks). When controlled by Logo, a Sail program (Sailogo in Figure 9) interprets commands to the train, remembers switch states and the train's block and direction, determines legal moves, prevents potential derailments, monitors the train's motion, and can find a path through any maze of possibly disabled switches (or announce that no path exists). The latter ability was not intended for students to use; students are supposed to write their own maze-solving programs.

In 1971 and 1972, prior to our receipt of BBN Logo, a Basic-like language and a small, traditional CAI curriculum were designed for and used by students to solve mazes with the train on the fixed layout. A schematic picture of the track layout is given in Figure 11. We use "+" to separate blocks, which are designated by numeral-letter pairs. Slashes ("/" or "\") cross other tracks at switches. Two "crossovers" (nonswitches) also exist in the layout (at locations "5B" and "3E"). A five minute color film, available from IMSSS, documents the project some time before Logo was modified to control the train.* The Logo/Train interface is also detailed in Appendix 2.2. Train commands are summarized there and in Table XI.

---

* We thank Steve Mylroie for his dedicated work on the hardware. The film was produced by Mike Raugh with the help of Marney Beard and Jonni Kanerva.

```
    G    F  E  D  C  B   A

6 -----+--+--+--+--+--+--    6
  !                    -
  !                   -
5 !  --+--+--+--+--+--+----- 5
  !       \      /  -      !
  !        \    /  -       !
4 +  --+--+--+--+--        ! 4
  !        /  /            !
  !       /  -             !
3 !  --+--+--+--+--+--+--  + 3
  !        -  /   \        !
  !       /  /     \       !
2 -----+--+--+--+--+--+--  ! 2
           /      \        !
          /        \       !
1 -----+--+--+--+--+--+----- 1

    G    F  E  D  C  B   A
```

Fig. 11. Schematic of the Logo-Controlled Train Layout.

In retrospect, and for the benefit of those who would also be interested in building this type of device, it is clear that precipitous construction of a fixed layout, any layout, is likely to be a mistake. We should first have graphically simulated the train domain, which in fact can now be done with the Logo-animation discussed earlier. Then, given the nature of the application, a hardware incarnation could have been selected. The existence of a simulation would also have allowed our work to continue, even when the hardware failed. Some of the value of a concrete, physical device is of course lost in any simulation, but the level of hardware reliability required of a real device is often underestimated. A model train is a particularly challenging piece of equipment in terms of reliability. Desirable abilities such as

Table XI

IMSSS Logo Train Commands

Name	Action
FRONT	move train forward a specific number of blocks
BACK	move train backward a specific number of blocks
HOME	move train to its starting location (see SETTRAIN)
TRMOVE	find a route and move train to a specific location
SPEED	set the train's speed
SETSWITCH	set the direction of a specific switch
SETTRAIN	set all switches straight, find train on track and put it at a specific starting block
CONNECT	join three specific blocks by throwing appropriate switches
TRAINFO	return information about the state of a specific block or switch
WHERETO	return a sentence of locations accessible from a specific location
WHERE	return a sentence of blocks under and to either side of train, and the state of any relevant switches
TROP	general Sailogo operator for communication with Logo (used for experimental commands)
WHISTLE	blow the train's whistle

independent control of multiple trains make heavy demands on any system for communicating between interface and rolling stock (engines and cars). A piece of dirt on the track can wreak havoc with naive train-monitoring schemes.

One reason why the IMSSS train has not been used extensively is its lack of reliability. In addition, the fixed layout and control facilities hamper the educational usefulness of the entire system. Ideally, students should be able to design their own layouts from a basic matrix by defining allowable connections. Then, problems of a graph-theoretic nature could be posed.

Evaluation of our implementation led us to believe that a generally suitable layout/simulation would have switches as nodes in a graph whose links are track sections. The switches would not be imbedded in blocks, as they are in our present physical layout, and a piece of rolling stock would not be allowed to stop on them. Nodes would be the center of activities like uncoupling as well as switching. In fact, a train simulation along these lines has been produced at another research facility by one of the authors. Users of that system can design their own cars, draw their own layouts and run as many trains as they please. In terms of the relative value of simulation, it is worth noting that the aforementioned simulation demanded an amount of effort some orders of magnitude less than that expended to build our physical system at IMSSS, and it is eminently modifiable.

Another type of device, which allowed students to make the computer "talk", was made accessible via Logo toward the end of the summer experiment. It is the digitized-audio system developed for and used by the Stanford reading project (Atkinson, Fletcher, Lindsay, Campbell and Barr, 1973). Using the Logo primitive 'SAY', students could make the computer utter sounds composed of any of 2000 prerecorded phrases, words, and phonemes stored on a system disc unit. As for most of the other devices we have discussed here, the audio system was accessed via Sailogo. During the fall of 1973, only one terminal with audio output was available to our students, on a limited basis. Nevertheless, several students produced guessing games, word games, amusing parodies of CAI like the Stanford reading program, narrations with deleted expletives, and a truly amazing program that could dial a telephone (via a special switchbox interface) and talk to whomever answered. In reality we have only added the aural equivalent of 'PRINT' to Logo, but when used in conjunction with graphics terminals, for example, students can attack problems like the coordination of dialogue and picture in movies. One student designed a talking turtle that narrates its drawings. If work with the audio device were to continue, we would like to give students the ability to record and access sounds that they produce themselves. For the interested reader, an improved audio recording and playback scheme is currently under development at IMSSS (Benbasset and Sanders, 1973).

3 Student Selection, Grouping and Tutoring

Our desire to draw some conclusions about programming languages and devices led us to design two linked sub-experiments (Table XII): children using teletypewriters (Groups I, II and III) and children using graphic displays (Groups IV and V). The first three groups would provide most of the data for comparing the languages, evaluating the curricula and characterizing tutor-student-machine interactions. It was hoped that comparing the behavior and performance of Groups I and IV

Table XII

Experimental Groups

Group	Composition
I	8 students learning Logo and then Simper
II	8 students learning Simper and then Logo
III	8 students learning Logo and Simper at once
IV	5 students learning Logo with graphics
V	10 paired students learning Logo with graphics

would suggest what advantages or disadvantages graphics has for novice programmers, while Groups IV and V might suggest how well students can work in cooperative programming situations. The graphic capabilities available to these groups were described in section 2.2.

Schools within bicycling distance of Stanford were contacted in order to obtain inexperienced volunteer programmers, 10- to 15-years old -- an age which is thought to ensure that children can master abstractions (Piaget, 1970).* Teachers and others recommending students were asked not to base their selections on students' performances in school, because we were interested in studying how any child learns to program. We had observed previously that teachers tend to recommend only their better mathematics students for such special projects. Apart from an admonition against such preference, we could not control the way in which the invitation "to learn how to use a computer" was presented to students, so we cannot be certain that our enrollees constituted a cross-section of local students. More students responded than were needed for the groups outlined in Table XII. We attempted to accommodate them all, including friends who appeared later during the body of the course. Figure 12 presents some information supplied by the enrolling students in response to a brief questionnaire. Since students typically heard about the course from their mathematics teachers, the preferences they expressed weren't surprising. In all, about fifty students involved themselves in the course at one time or another. To some degree, this insulated the experiment from the problem of dropouts.

---

* We are indebted to Carolyn Stauffer for her invaluable help as liaison.

One-hour classes were held in the mornings four days-a-week. In theory, Fridays were reserved for modifying the curricula and debugging the interpreters or devices. However, on demand of some of the more interested students, Friday was considered open too. Since we could provide no transportation, those students beyond bicycling range were transported by their parents. Parental inability to continue transportation created a few defacto dropouts.

In order to obtain an initial assessment of each student's aptitude for programming, and to point out possible problems that each student might later have in learning the concepts, we constructed a test consisting of questions gleaned from a wide range of sources. Unfortunately, we found no test in current use which impressed us as being valid for the range of concepts in Table I. A number of

Age/School Distribution

```
              me
              pe
              te
c             hv  la  st
h 10-         hv  me  lo
i             hv  hv  wo
l             hv  hv  gu
d             hv  hv  gu
r         te  hv  hv  ma
e 5-      te  hv  hv  ma
n         hv  hv  hv  ma
      fr  hv  hv  hv  ma  gu
      tr  hv  hv  hv  ma  gu
      hv  hv  hv  hv  ma  gu
   0- -----------------------
      10  11  12  13  14  15

              age
```

Age/Liking of School

```
                15
        14      15
        14  15  14
        14  14  13
        14  13  13
    14  14  13  12
    13  13  12  12
13  12  13  12  12
13  12  13  12  12
11  12  13  12  12
11  11  10  12  11
11  10  10  12  11
------------------
 1   2   3   4   5

dislike     like
```

Age/Subject Preferences

English

```
            15
15-         14
    14      14
    14      14
    14      13
    14      13
10- 13      13
    13      13
    12  15  12
    12  14  12  14
    12  14  12  13
5-  12  14  12  13
    12  13  12  12
    12  13  11  11  15
    11  12  11  10  13
    11  11  10  10  12
0-  ------------------
     1   2   3   4   5

dislike     like
```

Languages

```
    15
    14
14  13
14  13
14  13
14  13
14  12  15      15
14  12  14      13
13  12  14      12
13  11  14      12
13  11  13      12
12  11  13  13  12
12  11  12  12  12
10  10  11  12  11
------------------
 1   2   3   4   5
```

Mathematics

```
                15
                14
                14
                14
            15  13
            15  13
            14  13
        14  13  13
        14  13  12
        14  13  12
    14  13  12  12
14  14  13  12  12
14  12  13  12  12
13  12  12  11  11
11  12  12  11  10
10  12  11  11  10
------------------
 1   2   3   4   5
```

Science

```
                15
                14
                14
                14
                13
                13
            15  13
            15  13
            14  13
            14  13
            14  12
            14  12
            13  12
        13  13  12
14      12  13  12
13  14  12  12  11
12  14  12  12  11
12  12  11  12  10
11  11  10  11  10
------------------
 1   2   3   4   5
```

Fig. 12. Some Information Characterizing the Students.

commercial programming tests were examined and some questions from these were used. However, all these tests relied heavily on timed sections of multiple-choice, often repetitious questions.[*] Such structuring produces easily graded results and is commonly used to boost the "reliability" (correlation among test applications) of a test. We were inclined to place emphasis on the more elusive but crucial notion of validity.

A test, no matter how reliable, is utterly useless if it fails to measure the property of interest. It may even be dangerously misleading. In terms of the theory of testing, as currently applied in the social sciences (see Worthen and Sanders, 1973), validity like reliability is measured by correlative techniques. However, no matter how long the chain of correlations, validity is ultimately founded in human judgements and evaluations. While we intend no critique of testing theory and practice here, an example from one of the commercial test brochures is discussed in Appendix 3.1. It should alert the reader to some of the pitfalls that threaten those who wish to do aptitude testing, particularly with commercially available materials. The only conclusion we hope to imply with such an example is that testing theory and practice typically diverge when validity is demanded, and yet validity of measures is precisely what must be demanded when meaningful research is the goal.

---

[*] Tests included: the ARCO Computer Programmer, the CPAB and Flanagan Industrial Test series by SRA, the ECPI data-processing test, and the IBM programming aptitude test.

We had two purposes for presenting our new students with a test. First, a valid measure of a student's aptitude for learning the concepts would be needed for matched grouping of the students now (as per Table XII) and for a later study (Cannara, 1975). Second, we hoped it would be possible to match the way students attacked particular questions in the test with particular aspects of their performance in the course. Thus the test might be able to suggest the sort of tutorial help each student would need. The test itself was a subject of research. It was constructed of some questions taken from the commercial tests we examined and questions of our own design. All questions were formulated or reformulated to require constructive answers. A definitive portion of the test is reproduced in Appendix 3.2. For our purposes, multiple-choice items would be useless. We wanted to know what students thought about each question and why they gave their answers, even if the answers were wrong or incomplete. Detailed answers would help us evaluate the test as well as the students. Validation of the test, item by item, student by student, would be immediately subjective with no extraneous correlations.

About one-hundred questions were selected for possible use in the test. Before the questions were presented to the incoming students, we attempted to evaluate their difficulty and clarity, and the time required for their solution. We presented the entire assemblage to several programmers (children and adults) in the IMSSS community.* As a result of this simple evaluation, we decided to accept most of the questions and present them in two tests. About one-third of the questions were presented to students on the day they enrolled, and were to be completed in one hour. The remainder were to be completed at home at each student's convenience. The two parts of the test contained many similar questions. This strategy was chosen because the preliminary evaluation had suggested that time should not be a factor in the test. Thorough and accurate evaluation of both test and students seemed to demand that as many questions as possible be answered. The two-part testing would also suggest whether or not any time limit should be applied to the single test which would be used in the later study. Unfortunately, many of the students failed to complete the lengthy "take-home" portion of the test, either for lack of interest or because they left the course. Therefore, most of what we will discuss about the test will derive from results of the shorter, timed portion.

---

* We are grateful to Marney Beard, Doug Danforth, Adele Goldberg, Paul Hechinger, Greg Hinchliffe and Lauri Kanerva for their help.

We had selected questions according to their apparent value in testing the ability to manipulate unfamiliar languages, model or analyze processes, form deductions, and visualize figural transformations. Some of the questions proved to be very useful for discriminating among the enrolling students. Two of these, the "candy-machine" and the "numbers-in-boxes" problems, required an understanding of concepts directly related to programming. Errors made by the students on these two questions were especially interesting.

One of the questions presented a partial flow-diagram for a candy machine (the first problem in Appendix 3.2). A few states had been left blank and connections between some states were missing. The task was to complete the diagram in any reasonable way. Many students had trouble with the basic idea that a process can be represented on paper as a diagram of the sequence of events in the process. They left blank states empty, filled them inappropriately, or misconnected the dangling states. Errors in the solutions given could be divided into three classes: (1) assignment of unreasonable destinations for unconnected arrows, (2) assignment of unreasonable functions for undescribed states, and (3) treatment of the entire diagram as a maze in which only one path was to be marked as a likely protocol. Errors in class (1) or (2) suggested that a student had trouble using the information already present in the diagram to deduce reasonable "things to do next" or "things to do now". Class (3) is interesting because such errors indicated that a student viewed the diagram as instructions from which to choose one plausible sequence, rather than as a complete description of all possible sequences, for some process.

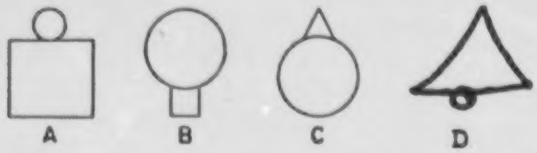
The other question (given as the second problem in Appendix 3.2) asked the students to obey a short sequence of arithmetic instructions which operated on some numbers in a set of numbered boxes. Very few students correctly obeyed the instruction which read: "Add the number in box 7 to the number found in the box whose box number is in box 6, and write the sum in box 6". The sentence is hard to read, but the idea that a number (value) in a box could be used as the number (name) of a box (indirect addressing) was the typical difficulty. Many students also had trouble with the idea that writing a new number into a box should destroy its previous contents. Solutions fell into a few distinct classes which can be attributed to failures in the understanding of those two concepts.

In a later section, we will discuss the relationship between students' solutions to test questions and their performance in the course. Our desire for constructive answers to all questions is best justified by those examples of "wrong" answers which nonetheless showed that students were thinking along the right lines. Figure 13 presents some answers for a question derived from a commercial programming aptitude test (note the subtle defects in drawings B and C, and the beguiling A-B sequence). It is important to note that answers like those in the figure evidence approaches to the questions which would have been counted completely right or wrong if nonconstructive answers (e.g. multiple-choice) had been required. Figure 14 shows examples of totally unexpected answers to a question of our own formulation. One can neither assess a student fairly, nor know what a test is testing (and therefore cannot begin to establish validity) if the scheme for generating answers critically warps or limits any information relevant to the purpose test.

Results of the test were used to establish a rank ordering of the enrolling students and then to assign them to experimental Groups I, II and III. Performance on the test seemed to break into a few levels, and roughly equal numbers of students from each level were assigned to the first three groups. Groups IV and V were formed according to student preferences on working alone or with a partner, so that no one would be forced into an uncomfortable situation. Groupings I, II and

The Question and the Desired Answer

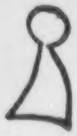
Figure A was changed into Figure B by a simple rule. Please draw figure D so that it corresponds to figure C changed by the same rule.



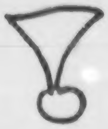
What is the rule in words? BOTTOM SHRINKS, TOP GROWS

Other Answers

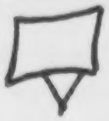
TURN IT UPSIDE DOWN AND ALTERNATE SIZE



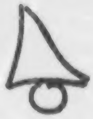
A IS A SQUARE WITH A CIRCLE, B IS JUST THE OPPOSITE



YOU CHANGE TO THE OPPOSITES



TAKE THE FIRST BASIC FIGURE AND CHANGE WITH THE SMALLER AND TURN UPSIDE DOWN



THE SMALL TOP FIGURE BECOMES LARGE AND THE OTHER BECOMES SMALL AND THEY TRADE PLACES

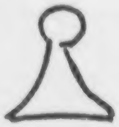


Fig. 13. Some "Wrong" Answers from the Preliminary Test.

The Question and the Desired Answer

What one rule, not using arithmetic, was used to make the digits on the right from the strings of digits on the left?

999999999	9
556	5
6106	6

TAKE THE FIRST DIGIT

Alternate, Unforeseen Answers

PREDOMINANT NUMBER

WHAT EVER NUMBER THERE IS MOST ON THE LEFT, PUT IT ON THE RIGHT

TAKE THE DIGIT WITH THE HIGHEST PLACE VALUE,
OR THE ONE THAT REPEATS MOST OFTEN

Note: "number" was acceptable although "digit" or "numeral" were technically correct. More than half of the students who gave complete answers to this problem seemed not to be aware of the distinction. Their rank and choices of words contrasted as:

student rank	"digit" or "numeral"	"number"
above median	9	8
below median	3	8

Fig. 14. Some Novel Answers from the Preliminary Test.

III were logical rather than physical because the students determined their own class schedule within the time constraints mentioned earlier. Scheduling of Groups IV and V was constrained to specific times because of the limited availability of graphics terminals.

Figure 15 shows the composition of the groups according to testing rank, age and amount of time spent in actual work with the interpreters. Notice from the table that the median tends to divide younger (10-12) from older (13-15) students. The candy-machine and the "logic" (the third problem in Appendix 3.2) problems tended to be most influential in discriminating among students of equal age above and below the median. The youngest had the most trouble with the candy machine. They missed the point that the diagram was an overall description of the machine. A few of the older students were familiar with flow-charts from school and thought that problem easy. They were also students who arrived after the course had begun. Late arrivals usually did very well with the test, perhaps in part because they could work on it quietly alone -- a feature lacking in our massed testing of the first enrollees.

Examining the test results in terms of four constituents, the first three problems in Appendix 3.2 and everything else, we can compare the students' performances generally as follows. Students at the bottom of the ranking were unable to grasp the candy-machine and the box-program questions, they correctly analyzed only the clearest statements in the logic problem, and they failed to finish the test by a large amount. Students near the middle filled only the empty states in the candy machine reasonably; they correctly obeyed all commands but the

	Group	Age	Hours Spent Using Logo & Simper
	III*$	15	36.4
	I$	15	35.7
	IV	12	23.3
++	V	13	23.8
+	II*$	15	18.1
+	IV	13	52.6
+	III$	13	29.7
+	III*	13	11.1
+	I*$	12	12.7
+	II*$	13	59.2
+	III#	14	0
+	I#	14	5.8
+	II	13	50.6
++	V*	13	33.4
	II	14	33.5
	II	12	22.8
	IV	11	22.7
	IV#	14	15.6
	IV	13	19.0
	II#	14	6.4
	II*#	14	27.9
	I#	14	5.7
	I#	14	5.7
median..	II#	14	0
	I	11	24.2
	II#	14	4.9
	III	11	14.2
	I	12	23.0
	I	14	18.0
	III*#	13	2.1
	III	12	11.0
++++	V#	12	4.7
++++++	V#	10	6.9
+ +	III	13	28.7
+ +	I$	12	27.1
++++++++	V	12	18.5
+ + + ++	V#	11	2.6
+ + + +	I*	10	11.7
+ + + ++	V#	12	13.5
+ + ++++	V#	12	0.9
+ +	II	12	21.1
+ +	III	12	20.1
+ +	I	12	19.1
+ ++++++	V#	12	9.4
+	III$	10	35.7
+	IV$	11	41.8
++++++++	V	11	25.0
	II	13	6.1

Legend

* marks students who enrolled after the course had begun.

+ joins graphics partners.

. marks significant breaks in performance on the test.

marks those who dropped the course early.

$ marks students who continued programming well after the course had ended.

Fig. 15. Student Ranking on the Preliminary Test.

the indirect-addressing command in the box program, with some failures to erase a box's content when they wrote into it; they only missed the fourth statement in the logic problem; and they did fairly well on the rest of the test, though not always finishing it. Students near the top correctly filled all states and connected all the dangling arrows in the candy machine, a few of them missed the indirect-addressing command in the box program, they did the logic problem correctly, and they typically finished the rest of the test. Of course this breakdown is not rigid. In particular, it is very hard to order many of the tests in the broad middle region of the ranking. Ranking forces transitivity upon performance ratings for solutions and problems which are often qualitatively different. But, however difficult our work was made by demanding constructive answers, the answers contained maximal information about the students and the test. If the test had been an exercise in multiple-choice, it is not clear what it would have told us, but it certainly would have told us less. Suggested changes in the test will be discussed later, as part of the experimental results.

We planned that the experiment depend upon written curricula which would control the basic information given to students. Interpreters for the programming languages would simply act as computational resources which the students could use to work problems in the curricula or experiment with on their own. However, we felt it was impossible to develop a fully self-contained curriculum for programming. In addition, our main concerns were gaining access to tutorial protocols generated by novice programmers while trying to give the students the best possible environment for learning. Therefore, we decided to provide human tutors who could help students over failures in the curricula and report to us on their interactions. The tutors were to be knowledgeable in the programming languages being taught and would be familiar with the corresponding curricula. We hoped to have enough tutors available each day to guarantee at least one for each five students in each group.* We emphasized two instructions to the tutors: (1) never type anything for the student on his or her own terminal, even when giving the most direct help, all typing must be the student's; and (2) when asked for help on any problem, encourage the student to formulate and try out his or her own ideas first, before making other suggestions. We hoped these instructions would guarantee the purity of the protocol data and help the students to think as much about generating and debugging ideas as about getting correct results. An evaluation of the tutoring effort will be included in our discussion of results.

4 Curricula

Development of "parallel" curricula for Simper and Logo proved to be the most demanding task in setting up the experiment. Both the concepts and the languages had to be taught, and this is done best with example problems, some of whose solutions students must copy, modify or generate. We felt our ability to teach both the concepts and the languages would be very sensitive to our choice of problems. And, for our students, we hoped that the course would serve to improve their literacy on the subject of computers and computation. Again our choice of examples and projects would be important.

---

* Our thanks go to Avron Barr, Marney Beard, Doug Danforth, Adele Goldberg, David Rogosa and John Shoch for their help as tutors

Unfortunately, documentation of problems used in similar work by others was scarce or cursory. Furthermore, most of the relevant research had been based on Logo or an equivalent high-level language. Problems appropriate for a low-level language such as Simper are typically quite different. That was the fundamental obstacle to achieving apparent parallelism, given the intentionally diverse natures of the languages to be taught. So, the curricula were constructed to teach the concepts in roughly the same order, using whatever features each language possessed that could best be exploited for each concept.

As well as the concepts, the mechanical details of each language had to be taught. A few features (line-editing) of Simper and Logo are very similar and were taught at the same time in the same way. But most features were taught differently, either because they were appropriate to different concepts or because they were needed at different times as tools in the general structure of each language. Appendices 4 and 5 supply glimpses of the Logo and Simper curricula as they were during the experiment. The Appendices and the discussion in this section do not reflect the changes to Logo, Simper and the curricula which resulted from the experiment.

Each curriculum was divided into logical parts (10 for Logo, 13 for Simper), each typically discussing more than one concept (Table XIII). Typically, these parts gave students programs to work on and fill-in-

Table XIII

Discussions of the Concepts in the Curricula

Concept	Logo Part	Simper Part
machine command language	1, 2	1, 2
alterable memory	2, 4, 5	2, 3, 8
literal	3	3
names and values	4	3, 8
evaluation, substitution	4, 5	3, 5
stored program execution	5	3, 4
decisions	8	5, 12
procedures	5	8, 11
procedure arguments	6, 7	7, 11
functions	6	7, 8
composition	6, 7	7
partial/total functions	7	7
context	4, 6	5, 11
changing context	7	11
recursion, iteration	5, 7, 8	4, 9, 11, 12, 13

the-blanks questions to answer. The parts were distributed one at a time, giving the tutors a chance to review each student's work on them. Those students learning Simper and Logo simultaneously (group III, Table XII) alternately received parts for each language. The concepts were presented roughly in the order of Table I. The concept of a heuristic was introduced via a scheme for thinking about recursive algorithms (Polya, 1957). This involved a brief case analysis of some problems:

(a) what case can be computed? (b) how do I detect that case? (c) if not that case, then how do I generate one closer to it? (d) what must I remember for each case? and (e) when do I stop? In procedural terms, (a) and (b) form the procedure body, (c) is the recursive step, (d) preserves local context, and (e) is the stopping rule.

A special effort was made to produce visually pleasing curricula. Path pointers gave direction to the student, making the next question or instruction contingent upon the student's latest response. This subtly introduced decision making and sequencing (program control). Cartoons and examples were chosen for humorous as well as conceptual merit, and frequent summaries were included so that the curricula could endure as reference material. Outlines of both curricula follow. Occasionally, it may be helpful to refer back to Sections 2.1 and 2.2 for details concerning the languages and devices.

Part 1 of the Logo and Simper curricula were identical and began with an informal discussion of Church's thesis and how it relates the potentials of human thought and machine computation. Some interesting capabilities of computers were illustrated. Part 2 used line-editing to illustrate that a machine can possess a memory that is alterable via commands in a simple, definitely nonhuman language. The substance of this part differed between the languages only to the extent that Logo has more line-editing commands (see Table V). Students were encouraged to type anything they desired, in order to test the very primitive error handling of the interpreters. The incompetence of many of the responses so generated was exploited to help students understand why present machines do not comprehend human languages (because humans do not yet understand how language is comprehended), and to tie this to Church's thesis and thinking in general.

From part 3 onward, the techniques for introducing concepts with Simper and Logo diverged. In the next subsections, we will discuss the remainder of the Logo curriculum, then that of Simper, and finally some differences between the two curricula.

4.1 Logo

Part 3 described Logo's literals (numerals and quoted strings) and the simple commands: 'PRINT' and 'TIME'. Turtle graphics students (groups IV and V) also tried simple graphics commands like: 'FRONT', 'LEFT' and 'PENDOWN'. This introduced Logo's left-to-right sequence of evaluation, as well as commands that return values. Part 4 dealt with name/value pairs, assigning to and finding values of names using either 'MAKE' and 'THING OF', or the colon notation (e.g. ':NAME:'). It ended with a short play which illustrated, via dialog, how a command composed of several operations and inputs is evaluated by Logo.

Part 5 directed students to copy and alter a procedure, 'RECTANGLE' (which drew (printed, on Teletypes$^{(R)}$) a picture of a rectangle and which students enjoyed modifying to draw a variety of other pictures, some censorable). The part presented an example procedure, 'TWORECTANGLES', that called on 'RECTANGLE' twice. Students in the graphics groups studied the same examples and solved many of the same string-oriented problems as the other students, but they also worked on procedures for drawing pictures. This exercised many of the concepts. We did not develop a truly separate graphics curriculum because the turtle domain does not provide many novel (in terms of the concepts) uses either for recursive procedures that return values or for conditionals -- beyond stopping rules for recursive drawings. Flow of procedure control was introduced in this part, as was simple recursion (the 'RECTANGLE' procedure calling itself).

Part 6 presented procedures with inputs and an output, the use of the 'TRACE' command for debugging, and analogies between procedures and functions with respect to composition and inverses -- for example, the two procedures:

```
TO DOUBLE :NUMBER:                    and   TO UNDOUBLE :NUMBER:
10 OUTPUT SUM :NUMBER: :NUMBER:             10 OUTPUT QUOTIENT :NUMBER: 2
END                                         END
```

which are nearly mutual inverses. Because 'UNDOUBLE' cannot handle odd numbers, it was cited as an example of a partial function on the integers. At this point, students had been exposed to sufficiently many concepts to be able to create significantly complex programs and make interesting (to us) errors.

Part 7 treated procedures which dealt with Logo's basic data-structure: strings. The following procedure was one correct solution to a problem derived from the preliminary test:

```
TO SWITCH13 :TEXT:
10 OUTPUT WORD THIRD :TEXT: WORD SECOND :TEXT:
          WORD FIRST :TEXT: BUTFIRST BUTFIRST BUTFIRST :TEXT:
```

-- it flips the first and third letters of an input word (procedures 'SECOND' and 'THIRD' had already been written as exercises). The solution served as an example of a function that is its own inverse. At the end of part 7, recursive procedure calls were presented as sequences of "little brothers" (Feurzeig et al., 1969; Brown and Rubinstein, 1973) with "knowledge clouds" describing their local environments. Students who were not helped by this were asked to consider a chain telephone call as an alternate analogy.

Part 8 dealt with decision making and the use of predicates, particularly in stopping rules for recursive procedures. It posed the following model of recursion:

```
TO CHOMP :WORD:                       _CHOMP "TAR"
10 TEST EMPTYP :WORD:                 TAR
20 IFTRUE STOP                        AR
30 PRINT :WORD:                       R
40 CHOMP BUTFIRST :WORD:              R
50 PRINT :WORD:                       AR
END                                   TAR
```

This model was chosen because it is not a "last-line" recursion (i.e. the recursive call on line 40 is followed by an affective command rather than by a stop) and requires the student to do some thinking about the state of the formal parameter (i.e. the value associated with "WORD" for each recursive call). The last parts of the Logo curriculum concentrated on problems to solve and projects to work on which required application of all the concepts. The nature and extent of difficulty students had with such projects could be used to judge the effectiveness of the Logo curriculum itself.

4.2 Simper

Part 3 of the Simper curriculum introduced the literals of the Simper language: decimal numerals. Names and values in machine language terms were also introduced. The part discussed the concept that a stored list of values is a program when it is executed by a machine for which those values have meaning. Part 4 motivated the sequential execution of instructions. Editing of memory locations illustrated another approach to the concept of alterable memory. Program control was introduced by a program which subverted (by writing into the program counter: register "P") the normal execution sequence. The program ran indefinitely.

That program was exploited further to illustrate the fact that the same algorithm often can be realized in more than one way. For example:

```
001 :PUT A 73           and         001 :PUT A 73
002 :PUT P 2                        002 :SUBTRACT P 3
003 :HALT                           003 :1
```

are computationally equivalent. The students enjoyed programs which ran on and on. A debugging feature in Simper allowed them to display registers and instructions as their programs were executed.

Part 5 attempted to clarify the three-level structure of Simper by contrasting the syntax and semantics of interpreter commands, assembler instructions and machine instructions. This would be a good place to treat the notion of computational context, since many students had trouble understanding that differing languages had to be used for communicating with the separate levels of Simper's structure. The part also introduced the decision-making operation: 'JUMP', and the notion of a program bug. The 'JUMP' operation offered a good test of a student's ability to predict what a given program would do. Students were encouraged to debug by pretending to be the Simper machine (see Berry, 1964). For particularly confused students, an egg-carton model of Simper's memory and registers proved helpful.

Part 6 classified Simper's assembly language instructions with respect to format and use. Special operations, such as 'ROTATE', were treated in detail, and new interpreter commands were introduced. The part acted primarily as a reference manual for operations and ASCII character codes. Part 7 reviewed the three essential characteristics of a computer (sufficient instruction set, accessible memory, controllable execution). The concept of a function was introduced using the character input/output instructions (i.e. 'CASK', 'CWRITE') which transform keyboard characters to/from decimal codes. This simultaneously introduced a new literal, the keyboard character, and the idea of computational context. Students seemed to have a lot of trouble grasping the latter.

The concept of functional composition and inverse followed naturally with a program which used the "B" register to link Simper operations which are mutual inverses:

```
001 :CASK B
002 :CWRITE B
```

Concepts of symmetry and domain could be introduced here because the above program cannot be executed backwards -- 'CWRITE' does not produce an output which is accessible to 'CASK'.

Students next worked on a program which realized a more complicated function (i.e.: X + X + 9), which was derived from the preliminary aptitude test. Partial functions were introduced using the 'ASK' operation, which accepts only numerals from the keyboard. The concept of a data structure was also provided by the character input/output operations, and by testing for arithmetic overflow or truncation. The latter could be exploited to illustrate non-determinism.

Part 8 introduced symbolic addresses (names) for memory locations. It pointed out that a name can be chosen to reflect the content of a location, making it easier to remember the name/value pair (but one student insisted that numerals were easier to recall). Students were asked to find an alternate realization for the function of the previous part (e.g.: 2X + 9) using names and, upon success, to synthesize a program that realized some function of their own choosing. Part 9 introduced relative addressing and data defined by a program which rotated five character codes into a single memory word. An inverse program rotated the codes back out, typing them on the terminal. Students enjoyed using these programs together to read in and print out some short words; and some, who were also Logo users, felt a new appreciation for Logo's facility with strings. This type of program offered many interesting debugging opportunities.

Part 10 dealt with indirect addressing, demonstrating again that the meaning of data depends on how and by whom it is used. A program which destroyed itself by decrementing an address used for storing was exploited to prove that the instructions understood by the underlying machine are simply numerals (the program could read its own instruction codes from the student, write them over itself, and keep on running). Indirect addressing was also used in a program that read a substitution code from the terminal and then translated "secret" messages. This helped to clarify what addresses are, it showed that programs and data are often segregated, and it introduced the "array" data-structure.

Part 11 formally introduced procedures and their calling sequences. Part 12 introduced stopping rules in an iterative procedure for typing dashed lines of any length. Part 13 merged the major programs in parts 9 and 12 into two linked procedures in order to define a new data structure: strings (the procedure was called "TYPE" in direct analogy to Logo's equivalent command). Students could load character codes into memory and print them out, thus making such things as posters possible, albeit tedious. Students were then asked to synthesize a procedure which created, anywhere in memory, a string typed from the keyboard. The final Simper part dealt with an implementation of recursive procedures using a pushdown stack to preserve local context. In the actual experiment, very few students reached this point.

4.3 Contrasts

The curricula (and the tutoring) were intended to teach computer literacy, especially in the sense that the computer is a very general tool for solving problems and that numerical processing has little to do with the principles of computation. The Logo and Simper curricula were, at least for this experiment, experimental. Ultimately, parallelism was sacrificed in favor of presenting the concepts via widely ranging applications of various features of the respective languages. Furthermore, pieces of the curricula were often synthesized "on the fly" if we found that what we had already written was not succeeding with the students.

Part 3 of both the Logo and Simper curricula opened by discussing the concept of a literal (numerals in Simper, quoted strings or numerals in Logo), but soon diverged. Simper students were taught that a machine's memory can be modifiable and observable, and that a set of values entered into it can be obeyed as instructions (a stored program). They necessarily were introduced to the Simper computer's structure of registers and memory. Bits of assembly and machine language were introduced along with a few interpreter commands (e.g. 'LIST'). Logo students, however, were not exposed to stored programming until Part 5. Here, they composed direct commands from simple operations (e.g. 'WORD') and literals, thus learning about Logo's string-oriented processing and learning that operations can pass messages among themselves. The concept of machine memory was treated only in terms of line-editing.

Logo Part 4 introduced name/value associations (via 'MAKE'), noise words, and a fill-in-the-blanks play which reviewed Logo's evaluation of a command composed of the few operations known so far to the students. Use of a value as a name (indirect addressing) was also introduced. For Simper, this was not discussed until Part 10, although name/value associations had been treated in Part 3. Simper Part 4 continued with the basics of programs, attempting to motivate the default order of execution (successor) and the ability to override it (by modifying the "P" register's content).

Simper Part 5 opened by attempting to clarify the tasks handled by Simper's three agents: the interpreter, the assembler and the machine. The students were asked to classify phrases in each of the corresponding languages. In spite of the slightly different interactions which are appropriate to command mode versus editing mode, this type of thing was not done for Logo. The Simper part also introduced decision-making in program control (using 'JUMP') and the idea of a "bug" (an unforseen error) which, in this case, caused a program to run forever.

For Logo, most decision-making was delayed until Part 8 and bugs were discussed first in Part 6. Logo Part 5 introduced procedures as both stored programs and new commands. Program control (in the sense of sequential procedure calls), program structure and simple recursion were motivated by drawing (or printing) multiple rectangles. The part ended with a rather complicated procedure that casually introduced decision-making to evaluate responses to a riddle. A short manual of commands and abbreviations, and a discussion of Logo's program-saving facilities were also included. Saving programs in files was now important to Logo students because they could write procedures that produced desirable results (pictures, etc.). Simper students would be introduced to program saving much later, when they could synthesize the relatively more complicated machine-language programs needed to produce comparable results.

Logo Part 6 began by discussing bugs and debugging and led into functions, inverses and composition. 'TRACE' was introduced as a way of spying on a procedure's true activity. Corresponding use of 'RUN' and the "ENTER" key had been made in Simper Part 3. Students were asked to synthesize and debug their own functions. Functions were not similarly treated in Simper until Part 7. Simper Part 6 began with a brief manual of machine operations and the ASCII character codes for future reference. It attempted to clarify the structure of assembly language for exceptional operations like 'SHIFT'. This led naturally into non-numerical processing of data. In contrast, Logo students began with such processing and did the most with numbers later, in Part 6. The part ended with a brief test of the students' understanding of Simper's tripartite structure, which had been observed to be particularly confusing to students.

Simper Part 7 reviewed Church's thesis in terms of the properties that a machine must possess if it is to be a computer. The Simper machine's characteristics were mapped onto this framework. Functions and related concepts were treated using character processing and the same visual analogies applied in Logo Part 6. Now Simper students were asked to synthesize their own functions. By making an inconsequential change to one function, they also saw that a function may be realized by more than one algorithm. The part ended by demonstrating the effect of finite word-length on the storage and processing of data (i.e., the effects of truncation and overflow on numerical data). Logo Part 7 paralleled the Simper discussion of computers. It then diverged and attempted to motivate the use of inputs to procedures by introducing new

Logo commands (e.g. 'BUTFIRST') and by suggesting several string-processing functions for students to write (e.g. 'SWITCH13'). A few "block and arrow" diagrams (flowcharts) were included to check the students' mastery of Logo's command evaluation and procedure execution. This type of aid was not used much in Simper. Logo Part 7 also suggested that a problem's solution could be structured by writing and then combining procedures which solve parts of the overall problem. This occurred much later in Simper (Part 11). The Logo part ended by discussing recursion again, using the "little-brothers" analogy, with and without inputs. The students were asked to synthesize a few recursive procedures, which proved to be a difficult task.

Logo Part 8 introduced decision making via the use of predicates and execution selectors (e.g. 'TEST' and 'IFTRUE'). 'PIGLATIN' and 'BINAR' (binary Game of Life) were exemplary applications. The use of decision-making in stopping rules for recursive procedures was also covered (e.g. 'CHOMP'). Simple, analogous applications of 'JUMP' had been made in Simper Part 5, but stopping rules only became important in the final Simper parts. Simper Part 8 introduced assembler symbols (names) for the purpose of making programs more readable. It also asked students to write more examples of functions and showed how bugs may appear even when applying simple arithmetic operations (e.g. division by zero). The latter was analogous to the treatment of 'DOUBLE' and 'UNDOUBLE' in Logo Part 6.

Simper Part 9 was concerned with further details of assembly-language addressing. It also introduced novel data-structures (e.g.

10-digit numerals as 5-letter words) by applying simple manipulations (e.g. 'ROTATE' and 'LOR'). Strange data-representation schemes were not discussed in Logo. Logo Part 9 continued to provide examples of recursive, string-manipulating procedures. 'MEMBERP' provided an example of how built-in data-structures could be exploited to represent special kinds of information, in this case sentences or words were viewed as sets. Along the same lines, students learned how to write procedures that could make letters print themselves as posters (using 'DO'); this was a response to obvious desires of most students. A similar concession was made in Simper Part 11.

Logo Part 10 consisted of many problems that could be solved by writing one or two recursive procedures. Students were encouraged to apply previously written procedures as tools (e.g. to use 'REVERSE' in writing a palindrome tester). A Morse-code problem analogous to one done in the same Simper part, and a graphics command interpreter for the turtle, extended the idea that the meaning of a message is ultimately defined by the recipient. Simper Part 10 introduced indirect addressing along with a potentially self-destructive program to carry the same point. Logo students knew just enough at this time to face the rather hard problems of Part 10. The Logo curriculum terminated here to allow students time to work on the part and on problems of their own choosing. Afterwards, interested Logo students could learn Simper.

The remaining Simper parts (11 through 13) attempted to motivate the use of procedures in machine-code programs, and to show how the equivalent of recursive Logo procedures with stopping rules could be realized in Simper. Hopefully from this, students would gain some appreciation for the inner structure of Logo and other high-level languages. Then Simper students could start learning Logo. The Logo curriculum did not discuss the structure of Logo itself.

5 Data Acquisition and Analysis

First, we will outline the simple methods we chose for obtaining data, given the experimental setup already described. Second, we will mention some features of the IMSSS time-sharing system which have had a negative influence on data acquisition or other aspects of this experiment. And third we will discuss the type of analysis we feel is appropriate for our essentially qualitative study and why that analysis cannot be founded naively upon classical statistical inference.

Throughout the experiment, the Simper and Logo interpreters saved information on each student's activities. Each command or response typed by a student was appended to his or her individual protocol file on the operating system's disc storage. Prompts and error messages elicited from the interpreters, and output from students' programs were also saved as they happened. Each such piece of information was tagged with its time of occurrence. From these files we intended to reconstruct each student's interaction with the languages and devices he or she used. At the end of the experiment, we modified the Logo and Simper interpreters to accept these files directly, in place of keyboard input. Each student's interactions with the interpreters could thus be replayed and be observed in their proper context. In addition, the error-message and timing data in the protocol files could be analyzed in more conventional ways by forming summary statistics such as error frequencies and typing delays (response latencies). This sort of data was not of particular interest to us except insofar as it could be used to point out particularly common errors, or confusions due to imperfections in the curricula or the tutoring. Some additional data were obtained from notes made by the tutors during the course of the experiment. However, as we will discuss later, the tutors were usually overburdened and were able to supply only a few comments on their interactions with the students. So, the bulk of the data from which results can be reported derives from replaying the automatically recorded protocols and recording our own work as tutors.

System Effects. Each day during the experiment, about forty students used the Logo and Simper interpreters. At any instant of time, between ten and fifteen students would be working. Ultimately, their interaction with the interpreters and our ability to obtain data were controlled by the operating system implemented on the IMSSS PDP-10. At this writing, that system is Tenex 1.31; during the experiment, it was Tenex 1.28, an earlier version. The Tenex system was developed at Bolt, Beranek & Newman Inc. of Boston under a U.S. Department of Defense ARPA contract to provide "virtual" memory management and other features to owners of PDP-10 machines. It is historically related to the SDS-940 time-sharing system. Certain aspects of the Tenex system, in either version, are at once elegant and troublesome. Some others are either misleadingly implemented, or logically consequent to the philosophy of the system's design yet implemented partially or not at all. We will discuss some examples. Only a few had detectable impact on the experiment. Our purpose is documentary -- for others who may use the same sort of system for similar purposes. The least significant system difficulties appear first.

(1) A few Logo and Simper operations (i.e. 'WAIT' and 'WAITM') have misleadingly inaccurate effect due to the nature of Tenex's program-scheduling algorithm. For example, the Logo command: "WAITM 60" may produce an execution delay ranging from sixty milliseconds to several seconds, depending upon the short-term system load. Unlike operating systems typically supplied by the PDP-10's manufacturer, Tenex does not advantageously reschedule programs which have dismissed their execution. Thus, from a program's point of view, a dismissal interval can be specified only by a nondeterministic lower bound.

(2) Also pertinent to scheduling, and relevant only to our design of the Logo/Sailogo system for operating various devices, are delays imposed by inter-"fork" communication. Tenex forks are pseudo-parallel, superior/inferior program contexts which may be set up as parts of one user's job. Each job is allowed a maximum slice of processor time whenever it is ready to run. Thus Logo and Sailogo communicate and run sequentially, but as one job. One might therefore expect that any time remaining in a Logo job's time-slice, after Logo has sent a message to Sailogo, would be available to Sailogo to carry on the computation. This is not so, because Tenex considers such fork communication as an input-output wait and reschedules the Sailogo fork. The same is true for communication in the reverse direction. Depending

upon the instantaneous system load, such "invisible" rescheduling can cause abnormally long delays even in simple Logo/Sailogo interactions. In our experiment, this effect was most seriously felt when students were doing turtle-graphics drawing and animating. For instance, after the experiment, when Simper was modified to contain, in the same fork, the graphics portion of Sailogo (easily possible because both Simper and Sailogo are Sail programs), Simper produced turtle drawings roughly twice as quickly as did Logo.

(3) Another problem arose when we attempted to simulate the control of one device by more than one student. The real train could only operate one engine and respond to only one controlling program, so we attempted to create a situation in which at least two students could interact with a single, multiple-train layout in a cooperative problem-solving situation. The most straightforward design should be the linking of two Logo jobs to the same simulation by reading from and writing into one another's memory space. One of the allegedly elegant features of Tenex is that every information handling entity in the system mimics the behavior of a file (even terminals are viewed as readable/writable files). However, one user's job cannot access another's fast-memory space, although jobs can share and communicate (more slowly) via disc-storage files. Thus forced to design a one-job, multi-fork simulation (one Logo fork per user, with the main simulation controlled by a superior fork), we found that Tenex would not allow multiple primary terminals for one job. The pseudo-interrupt system required by Logo (e.g. to allow "control-G" to function) could not be enabled for more than one student user. Success might be achieved by

someone more familiar with the bowels of Tenex, but we and the system-maintenance staff at IMSSS failed. Therefore, we could not make the multiple-train simulation part of our experiment.

(4) The nature of Tenex's disc-file management influenced our ability to save individual protocols. Given the number of students enrolled in our course, we had to maintain at least one-hundred distinct files for immediate access by Logo and somewhat more for Simper. Typically, half of these were protocols, the remainder student programs. Unfortunately, Tenex 1.28 provided for a maximum of about 120 files per directory -- when a directory is full no new files can be created until some are expunged. In such a circumstance, no new protocols and no new student programs could be saved. We considered saving all protocols on one file; this would greatly reduce the chances of saturating a Tenex directory. The idea was to append data continuously, in what is termed "thawed access". This simply means that more than one user may write or read the same file. However, thawed access proved useless because only data for the last student to close such a file would indeed be saved. To our knowledge, this potentially useful feature has yet to be correctly implemented in any version of Tenex. The present version (1.31) at IMSSS eases the directory saturation problem because it almost doubles the allowed directory space. However, a more elegant solution, based upon the dynamic directory allocation schemes of earlier PDP-10 systems, would offer permanent relief. A related difficulty stems from the lack of full "device-independent IO" in Tenex. One cannot randomly pick two system devices (e.g. a DECTAPE$^{(R)}$ and a standard magnetic tape) and send data uniformly from one to the other. Only some connections

can be made without recourse to special programs. This only influenced our weekly saving of student data, for which a special program was used to create directories and write files onto standard magnetic tapes. Again, this is an example of an effort which would have been unnecessary had Tenex's designers incorporated certain important abilities of earlier PDP-10 operating systems.

(5) Our final comments pertain as much to what the IMSSS time-sharing system must do, and should be, as to what Tenex is. The nature of a "demand-paging" system such as Tenex is to break all programs (and files) into uniform "pages" (blocks) of information. Since today's computer technology puts a premium price on fast memory, most systems, including that at IMSSS, have insufficient resources to allow all active programs to reside in fast memory. Thus a paging system uses not-so-fast, inexpensive backup-storage to expand a machine's apparent memory space. Each user has virtually the machine's entire memory to work with, but is subjected to transfer delays and rescheduling whenever his or her program demands access to pages not in fast memory. Theoretically, this allows diverse uses of the machine in an efficient, time-sharing mode (see Denning, 1970). However, at IMSSS, the bulk of daytime computation has been in shared (reentrant) programs. For example, students using Logo share large blocks of the interpreter. Only data pertinent to each student varies. Yet, in busy periods when many users demand memory, a paging system like Tenex swaps onto backup storage even the frequently used shared pages of student programs. The fact that even the smallest, one-page programs undergo such blind shuttling accounted for some of the poor performance of Tenex when it

was first implemented at IMSSS and was confronted with loads, typical of CAI, generated by scores of students all running in a few, small programs. For our experiment (even more so for traditional CAI), it would have been useful to have been able to "lock" small, reentrant programs into fast memory, thus reducing the paging load while usurping a relatively small fraction of available memory. Such an option is not conveniently available in Tenex (it is under belated study at IMSSS). It should certainly be considered by anyone intending to use analogous paging systems for simple educational programming loads in which fast response to interactions is paramount. As did the train, Tenex became a "fait accompli" at IMSSS without a deep concern for, or a thorough preliminary analysis of, its real-life behavior.

These comments have concerned aberrations in the IMSSS time-sharing system which might influence the service students receive as they work. Hardware problems, mainly stemming from parity errors in the relatively unreliable memory in use (to this date) at IMSSS, also affect its users. No system can always recover from such errors and often must be reloaded, destroying all users' programming contexts. Some data were lost in this way, and, more seriously, many students lost their programs, twenty or so minutes of their sessions and their rhythm with the curricula. Other problems, which do not generally affect student activities, will not be mentioned here. In general, only (2), (4) and the memory errors mentioned above presented daily nuisances to our experiment.

Analysis. The stated purpose of this experiment turns upon our ability to understand our students as they have tried to learn Logo, Simper and the concepts explained in the curricula. We are not concerned with classical hypothesis testing, although others have attempted to reduce their analyses of children learning programming to clinical forms, e.g.

> "Children who have had a Logo experience for several semesters will perform significantly better on problem solving tasks than children who have been in a non-LOGO control environment.
>
> Scores on a test for recursion in daily communication by children who have had a LOGO experience for several semesters will be related to their ability to use recursion in LOGO programming." (Folk et al, 1973)

Our goal has been the exposure of basic features of how children think in the relatively unconstrained environment of a programming laboratory. That is a qualitative exercise, and it centers on a detailed study of errors made by students as they try out new ideas for themselves. But, an analysis of errors must be valid in the sense that the essence of their protocol is not warped by analytical constraints. Whenever statistical procedures (such as classical hypothesis testing) are applied to data, certain mathematical assumptions (of scale and distribution) about that data must be met. In too many educational settings, the importance of these assumptions is ignored. Yet arbitrary assumptions lead to technically invalid or misleading analyses. We will discuss some common statistical pitfalls in more detail after we demonstrate how we have analyzed errors recorded in students' protocols.

An example taken from Simper protocol data illustrates the nature of our analysis. The example shows how one student suddenly seemed to grasp a concept with which he had been having trouble -- name-value association (addressing) in Simper. If the programming is unclear, the reader should refer back to Section 2.1, keeping in mind that a modified Simper is described there. The student's dialog with Simper is reproduced here as he was engaged in writing a program to realize the function: $x^2 - 3$:

```
003 :2

015 :ASK A
016 :STORE A 200
017 :MULTIPLY A A
018 :SUBTRACT A 3
019 :WRITE A
020 :RUN 15/
```

He appears to understand the purpose of addressing in 'STORE A 200', but his program contains several errors that suggest otherwise. The first causes execution to stop at 017 because the symbol "A", used in the address field of the instruction in 017, has no binding and thus no associated value. The student thought he could square the A register's content with the instruction: 'MULTIPLY A A', and he thought he could subtract 3 from that with: 'SUBTRACT A 3'. In both cases, the meaning of the register field seems to be understood, but the address field is misunderstood. The student corrects the first error (messages from the interpreter are in lower-case):

```
020 :FIX 17
017 :MULTIPLY 200 200
200 isn't a register, use a, b, or p
017 :MULTIPLY A 200
020 :RUN 15
```

and the program works except that, because location 3 contains the value 2, the subtraction doesn't do what he expected. At this point he seems to understand that he can store and access values via addresses (names) because of his correct use of the register and address fields of the 'STORE' and 'SUBTRACT' instructions. But the idea crystallizes:

```
020 :FIX 201
201 :3
```

when he associates the desired value 3 with the name (location) 201,

```
020 :FIX 18
018 :SUBTRACT A 201
```

and correctly accesses it to complete his program. From this dialog, one can see the student begin to apply the concept in correct fashion (in the 'STORE' instruction), then fail because he has not yet mastered it fully, and finally succeed, partly helped by simple error diagnostics. The student later made a similar mistake, but corrected it at once.

For the purposes of the experiment, this type of analysis can suggest when and how a student masters something presented in the curricula. Students can be compared in far greater detail than can be achieved with discrete tests, the curricula and languages may be evaluated very finely, and the preliminary aptitude test's validity may be rated subjectively.

Protocol analysis also allows us to evaluate the languages by showing us how they help or confuse novice programmers. The following are brief examples from Logo and Simper protocols of absurd or misleading responses to syntactic errors. First, consider:

```
_PRINT :::SNOOPY:::
don't use the empty thing for a name
```

in which the student's obvious attempt at multiple indirect-addressing is completely misconstrued by Logo's simplistic parsing (the first pair of colons are found to contain no name string). And second:

```
001 :SUBTRACT 1 FROM P
002 :RUN
warning! you forgot to name a location fromp
illegal memory reference 0 at 1
```

in which Simper, striving to extract three fields and no more from the student's line, compressed a simple syntactic error and generated a more advanced type of error. Not only was this spurious error unrelated to what the student had done, it exposed the student to a situation for which he was not yet prepared (i.e. the use of assembler symbols).

Examples like those can be used to guide language design. Since the experiment discussed in this report was in part a pilot study for a later experiment, our protocol studies led to changes in Logo and Simper in preparation for that experiment.

It should be clear that the interpreters we have used are not "smart". They do not tutor their users on the semantics of programs -- in the experiment, that was left to humans. The interpreters do little more than trap syntactic errors, sometimes acceptably well:

```
001 :SHIFT
unspecified register, use a, b, or p
001 :SHIFT 76
76 isn't a register, use a, b, or p
001 :SHIFT A
shift uses l, or r or @ and a number in the address field
001 :SHIFT @56
@56 isn't a register, use a, b, or p
001 :SHIFT L 56
l isn't a register, use a, b, or p
001 :SHIFT A L57
```

As we mentioned earlier, a simple analysis of the protocol files was also carried out (e.g. Figure 16) to provide us with a few summary statistics which might point to difficult areas of the curricula or give us a very coarse measure of student performance. For example, if a Simper student's errors were categorized and plotted as in the graph in Figure 16, an interesting effect usually could be observed: familiarization with the language led to a decrease in errors classed as syntactic and an increase in those classed as semantic. We infer that as students increase their active programming vocabulary, they can more easily realize their ideas about problems as programs and find that their ideas (now programs) aren't always debugged. This is more reasonably corroborated by tutorial data and detailed protocol analysis.

We now return to our general discussion of the care which must be, yet often is not, exercised in applying classical statistical techniques to the analysis of data like those generated by our experiment.

> "It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts." (Sherlock Holmes, by Sir Arthur Conan Doyle)

```
bill163.dta;3  AUGUST 1, 1973 12:20PM

1 DAYS, 1 LOGINS, 33.40 MINUTES ON, 372 KEYS TYPED ON 60 LINES.

RESPONSE DELAY, MEAN & DEVIATION:  32.15  34.36 SEC.

 1.00 LOGINS/DAY, 33.40 MINUTES/DAY, 372.00 KEYS/DAY

 33.40 MINUTES/LOGIN, 372.00 KEYS/LOGIN, 60.00 LINES/LOGIN

 11.14 KEYS/MINUTE, 6.20 KEYS/LINE, 1.80 LINES/MINUTE

36 ERRORS:  36 GENERAL, 0 NAME, 0 RUN, 0 FIXUPS

36 SYNTAX ERRORS, .60 SYNTAX ERRORS/LINE, 1.08 SYNTAX ERRORS/MINUTE

 .00 RUN ERRORS/LINE, .00 RUN ERRORS/MINUTE

  0        10        20        30        40        50        60
  |    .    |    .    |    .    |    .    |    .    |    .    |    .
1|###################
2|##########
3|##
4|#
5|#
6|#
7|#
8|#

1|UNSPECIFIED REGISTER, USE A, B, OR P
2|EMPTY ADDRESS FIELD?
3|SHIFT & ROTATE USE L, R OR @ & A NUMBER IN THE ADDRESS FIELD
4|EXCHANGE USES A REGISTER IN THE ADDRESS FIELD
5|ONLY VALUES FROM 0 TO 999 MAY BE PUT
6|¬ ISN'T A REGISTER, USE A, B, OR P
7|SHIFTS OR ROTATES MUST BE BETWEEN -999 & +999
8|UNKNOWN OPERATION ¬
```

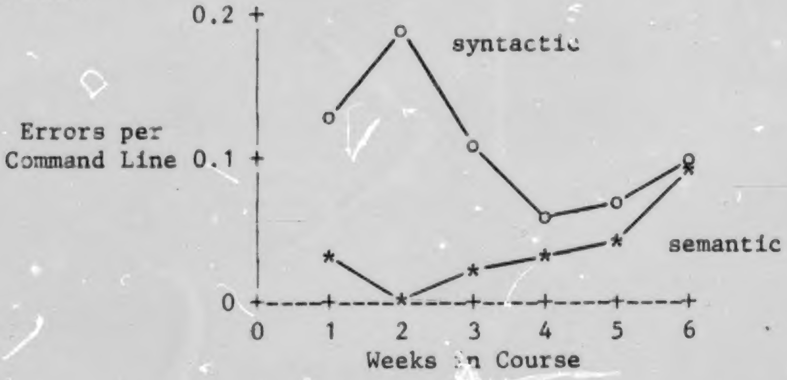


Fig. 16. A Simple Quantitative Analysis of Protocols.

Conan Doyle and his Holmes were not statisticians, but the quoted remark is nevertheless apt. In the social sciences, especially in education, the style of research too often reflects a Quixotic quest for numerical results, apparently stemming from the belief that quantitativeness is a precursor of objectivity and respectability in one's discipline.

> "They use statistics as a drunkard uses lampposts,
> for support rather than illumination." (Andrew Lang)

(the reader is welcome to replace "statistics" with "references" or "quotations"). This is amplified by the relatively easy access most researchers now have to computerized statistical procedures (Ellis, 1972). Perhaps more seriously, widespread use of standardized procedures has led to stereotyped theorizing (e.g. to hypothesis testing restricted to linear models and Normal distribution theory -- procedure defines theory), while the implicit assumptions of the procedures are virtually ignored.

Consider, for example, our preliminary test (Section 3) for "programming aptitude", whatever that may be. We could score the results of students' work on it on some scale, say 0 to 100, to get a list of numbers that could, perhaps along with other numerical data, be injected into a vast number of standard statistical programs. We could compute correlation and regression coefficients, and test hypotheses. But, the relevant statistical procedures have been derived from a set of first principles and assumptions. What about them? Mustn't all the restrictions they imply be met reasonably well by our data and the scoring process? Are the theories defined by these procedures relevant to the kinds of questions we wish to ask about, and will they shed light upon, the real-life process that produced the data?

Take for instance the word "scale", used in describing how we might produce "scores" for our students. Many (educational) researchers believe that tests define scales. But a scale of measurement is a well-defined mathematical object. It is an abstraction of a particular property of other objects. Every device (test) that purports to generate a scale must, apart from truly measuring some property, produce results which do not violate the principles which define a scale. For instance, to be represented on an interval scale (as points on the number line), data must have (among others) the property of transitivity of point and interval. In other words, any data points a, b and c obeying, for instance, the relation: $a > b > c$ must also obey: $a > c$; similarly, any intervals p-q, r-s and t-u obeying: $p-q > r-s > t-u$ must also obey: $p-q > t-u$; and so on, for other relations. Valid test scores cannot be taken as interval-scale data unless all scores which differ by equal numerical amounts imply equal differences in amount of the property measured by the test. So, a test must be uniformly valid. Consider the much-maligned "IQ" (standardized intelligence) test. Students who score between 50 and 60 must differ respectively and by exactly as much in whatever the test measures as do those who score between 120 and 130, and so on, for all possible differential scores, otherwise the score data are at most of ordinal significance. Unfortunately, such data are often reported as interval data. This can be, for at least three reasons, misleading or plainly wrong.

First, students who score low on a test do so because they know how to answer few questions, while students who score high may answer the low-scorers' questions and others as well. Different questions (answered by different students) alledgedly test different abilities, intentionally so. Tests are often divided into subtests which reflect the theoretically diverse abilities to be tested. But combining scores from all questions on a test into one descriptor may negate the assumption that equal score intervals are commensurate. Unless all questions or subtests can be shown to be disjoint and distinct _in the same way_ for _every_ student who takes the test, reporting one summary statistic per student and presenting a distribution is folly. An inch at the low end of a yardstick measures as well and means the same thing as an inch at the high end. Can the same be said for IQ or many other mental tests? Saying that such tests "measure what they measure" skirts fundamental mathematical issues of data representation. They are not analogues of yardsticks.

Second, a measuring instrument which interacts with the individuals it measures may not produce even ordinal data. Consider the cultural bias which various tests are said to impose upon the testee -- an allegation that has recently been sustained in some courts. Tests which ask humans to think will observe human thinking, subject to the vagaries of human psychology. How useful is a yardstick that is highly and ambiguously sensitive to heat, light and the day of the week?

Third and last, even if a test validly represents its results as interval data, those data are often "standardized" by transformation into a convenient distributional form, often the Gaussian (Normal), which may have nothing whatever to do with the population process that gave rise to the data. Such preprocessing ("transgeneration") is common to many computerized statistical procedures, and can be legitimate. But, for instance, IQ data is manipulated by hook or crook to fit a Gaussian form. Obviously, at most ordinality is preserved, and the resulting "bell-curve" is totally without redeeming scientific importance. Forcibly molding data to fit analytical constraints can destroy the meaning of both the data and the analysis. Of what use is a "silly-putty" yardstick, perhaps grading from meters to fathoms to feet, if the questions we ask of its measurements cannot properly account for its latest non-linear form?

Apparently, we do not understand the psychology of testing (or learning) well enough to fairly represent mental-test results as more than ordinal data. But analyses derived from distribution theory, like regression, correlation, and analysis of variance and covariance, assume that the data they operate upon is at least interval-scale data. And such procedures are applied daily to mental-test scores. What can be inferred from such misapplications? Mathematically and scientifically, nothing. Blind use of statistical procedures can only give a false sense of objectivity. One does not have a scientific result when one applies an analytical technique in violation of any assumptions on which that technique was derived, unless one also has a theory which describes the perturbations induced by each and every violation ("robustness" and the "law of large numbers" don't so qualify). One seems to have results, because procedures can't judge the source and calibre of their numerical input -- cube roots of license-plate numerals observed in travelling cross-country will feed most procedures, but what do they measure? Any act of measurement, physical or social, must be done in the light of a theory of how the measurement interacts with the measured. Not a trivial theory (e.g. of signal-plus-Gaussian-noise as in classical test theory), but a theory that defines the signal (and scale axioms) in relation to other objects. Yardsticks and radar can abstract the same property of objects, but it is the same property because the theories of the two devices mesh. Statistics enters only as a "theory of errors", when we wish to judge "noisy" measurements in some organized way; the noise is ancillary to, yet affects not the theory of, the signal. But variations in human behavior (e.g. as "measured" in educational testing) are not always noise to be described away by statistical conveniences like analysis of variance. Likely as not they are data whose meaning might point to reasonable educational theories that have eluded simplistic analyses, or, sampling from Reneé Haynes' postscript for Koestler (1973):

> "This matter of quality as contrasted with measurement ... seems to me to emerge with ever-increasing urgency. It cannot be ignored simply because it is so uncomfortable and so difficult to deal with. It is relevant to science ... Yet (because it is so much easier to accumulate and to quantify data than to reflect on their significance) quality and meaning, which matter most to men, tend to be brushed aside."

Research that applies "damn-the-torpedoes" quantitativeness produces no results, because the link between the data and the analysis has been broken, for instance, by vacuous scaling. Yet there might have been results. Many valid analyses can be carried out on a body of data without the need for cavalier assumptions about its structure (e.g. see Bradley, 1968 or Puri, 1970). The analysis should fit the data, not the other way around, to paraphrase Sherlock Holmes.

Misuse of scales is only one criticism that can be levied against free-wheeling statistics. Here, we will not discuss others, which range from data-"improvement" techniques like "Windsorizing" to confusions of deduction and induction. Let us return to the example of our own test.

The preliminary aptitude test's results were presented as a rank-ordering of our students (Figure 15) obtained by a "forced-choice" evaluation of their work. Perhaps even this is not justifiable, for a test whose validity has yet to be determined by experimental results. At least a few students, especially near the median, might well be reordered or considered hopelessly tied. Yet rank-ordering enforces transitivity. The theory behind our test is simple and qualitative: take as questions examples of the thinking that programmers are typically asked to do, where some types of thinking are more important, in the programming sense, than others. The former relates to validity, the latter to transitivity. No part of the theory suggests cardination or interval scaling. We feel that a careful, subjective evaluation of constructive answers produced by students can more nearly approximate an objective technique (if one exists) for ranking them than can a falsely objective testing/scoring procedure. Our theory may be wrong or incomplete, but determining that is one purpose of the experiment: what do students' interactions with the test have to do with their interactions with the course and with programming? Again, theory is eternally subject to data. The test's initial validity teeters on our subjective choice of questions, and it will stand or fall depending upon experimental results. Surprisingly many mental tests go unevaluated by their users who nonetheless report results of their use (e.g. see Folk et al. 1973 -- although we have been critical of their application of statistics, their report is otherwise valuable).

Why is misuse of statistical procedures common, perhaps growing? Apparently because many believe that social/psychological research can only be substantive if it mimics the quantitativeness of the physical sciences. Ultimately, dispelling this rationale is the responsibility of statistics teachers (especially those who teach, or are, nonstatisticians), who must instill not only broader knowledge, but a sense of responsibility and respect for the use of statistical inference in decision making and theory building.*

6 Results

Virtually every component of the experiment was evaluated in some way. The students provided feedback both directly and indirectly, by supplying specific opinions to us and by our observations of their behavior and attitudes. Figure 17 summarizes the students' responses to a questionaire they received from us shortly after the experiment had terminated. The total numbers of opinions for all rows are not

---

* We are indebted to Mario Zanotti for sharing with us his knowledge of the foundations of mathematics and statistical inference.

	Tone of Student Remarks		
Subject	Negative	Noncommittal	Positive
Plotter		1	16
Graphics Turtle		2	26
Games		3	25
Tutors	2	3	25
Return again	2	3	25
Train		3	14
Robot Turtle	1	1	9
Logo		8	21
Logo Lessons	3	8	18
Simper Lessons		5	5
Simper	3	3	5
Teletypes$^{(R)}$	4	12	11

Subjects are ranked on relative fraction of positive remarks.

Fig. 17. Student Preferences.

identical because some students responded partially and others were not sufficiently exposed to every item to render an opinion.

Most of the preferences expressed in Figure 17 correlate with casual comments made by the students during the experiment. For instance, the plotter was preferred to the robot because "it draws better" (it produces more faithful drawings). The plotter was preferred to the IMLAC$^{(R)}$ graphics because it produces portable, permanent results, and because one can "see it work" -- this fascination might have worn off had we had the plotter longer. Graphics was preferred to the robot because it was faster, more accurate, and personally available for each student. Animation could not influence these opinions because it was not available until the very end of the experiment.

The item listed as "games" in Figure 17 refers to certain programs accessible to students on the IMSSS system, such as Hangman, which were intentionally not announced until the students completed most of the curricula. Some students, of course, accidentally discovered a game or two. Our policy was that games could be used after a student's daily session with Logo or Simper. The most popular games were: highly interactive, like Hangman; those involving more than one player, like Poker; and those with plenty of action, like Spacewar. Wordy, random-number-driven games, like Football, were thought "dumb"; unless they tickled a specific interest, as Startrek often seems to do. Games were ranked to give us an idea of their place in the students' view of the experiment. We encouraged students to write their own games and used some as examples in the curricula.

One prevalent opinion among students familiar with both Logo and Simper was that "it's harder to do things in Simper". This resulted in most students preferring to work with Logo, regardless of the starting language. Figure 18 tabulates the proportion of time students spent using Logo (and, by complementation, spent using Simper) for all non-graphics students. Note that, within each group, students are ordered

(Logo hours / Simper + Logo hours, versus pretest rank,
"-" denotes students who took the test but not the course)

Group I

```
 .99    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .98    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .99    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
```

Group II

```
 .70    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .34    XXXXXXXXXXXXXXXXX
 .48    XXXXXXXXXXXXXXXXXXXXXXXX
 .22    XXXXXXXXXXX
 .31    XXXXXXXXXXXXXXXx
0.0
 .17    XXXXXXXXx
  -
0.0
 .04    XX
0.0
```

Group III

```
 .82    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .69    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .64    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  -
 .87    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .67    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .69    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .68    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .68    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .88    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Fig. 18. Breakdown of Students' Programming Time.

by pretest rank. Thus Figure 18 may be correlated with Figure 15 to obtain further information. This convention will be observed in other figures in this section, whenever it is appropriate.

Because few students finished (reached the last part of) the Logo curriculum, Group I spent negligible time with Simper. But many Group II students went far enough with Simper to be able to start Logo, partly motivated by their seeing their friends' work. What is interesting about Group II's behavior is that the students who began using Logo stayed with it, virtually to the exclusion of further work with Simper. Figure 18 also shows that students given simultaneous access to Simper and Logo (Group III), and subject only to the stricture that Simper and Logo curricula parts alternated, chose to spend most of their time with Logo. This group answered a capability question: students can learn both languages, nearly simultaneously, and do so faster than students who learn the same languages sequentially.

Mass preference of Logo to Simper was, in our view, a desirable outcome in terms of the students' computer literacy. Although Simper provides a convenient way to learn and experiment with assembly/machine language programming, students could see the advantage of a high-level language. Logo offers what students seem to want: easy access to message and picture processing. It offers a computationally more important feature: ease of phrasing complicated control structures. However, we found that appreciation of this latter idea was usually confined to the more able students.

Before further discussing the students' behavior, let us consider gross aspects of what the experiment suggested concerning the validity of the preliminary test. For Group II, Figure 18 indicates a correlation between students' ranks on the pretest and the time they needed to complete the bulk of the Simper curriculum. Because of the general desire to use Logo and our inability to distinguish (from summary protocol analysis) personally-motivated use from curriculum-motivated use, the data for the other groups do not relate to pretest rank. Students were also ranked by us and the tutors according to programming ability and dedication to the tasks presented to them in the curricula. Figure 19 shows these ratings, again by pretest rank, for all Simper students.

Figure 19 also tabulates the mean rate of errors in each student's commands throughout his or her work with Simper. Some slight, joint trend of error rate and pretest rank seems evident. For example, the median rate (.11) for those above median rank is much lower than the median rate (.26) for those below median rank. The reader can easily find other indicators of asymmetry in this sample of data that suggest a positive correlation between rank and error rate. However, we must caution that averaging errors in this way blurs the nature and importance of individual errors. Without referring to detailed protocol analysis, such a correlation merits little more than a "that's nice". We should mention that typing and reading ability varied greatly among the students. Furthermore, some students forged along, not caring how many errors they made, while others worried inordinately about making mistakes, particularly observed ones. Various

Groups II and III (Simper data)

("-" denotes students who worked less than 3 hours)

		Rankings Based Upon Subjective Evaluation of Performance	
Errors per Command		Mastery	Perseverance
.06	XXX	1	3
.14	XXXXXXX	3	2
.11	XXXXXx	3	3
.26	XXXXXXXXXXXXX	4	3
.03	Xx	2	1
-		-	-
.07	XXXx	3	2
.07	XXXx	2	1
.16	XXXXXXXX	4	1
.34	XXXXXXXXXXXXXXXXX	5	4
.23	XXXXXXXXXXXx	4	4
-		-	-
.26	XXXXXXXXXXXXX	5	4
.50	XXXXXXXXXXXXXXXXXXXXXXXXX	5	5
-		-	-
.26	XXXXXXXXXXXXX	4	4
.15	XXXXXXXx	6	5
.12	XXXXXXx	6	2
.16	XXXXXXXX	5	4
.34	XXXXXXXXXXXXXXXXX	5	4
.27	XXXXXXXXXXXXXx	6	3

Fig. 19. Simper Students' Performance Versus Pretest Rank.

combinations of such abilities and attitudes obviously can confuse simple comparisons of error rates. It happens that the fourth-ranked student (Figure 19, with a high error-rate) fell into the "unbridled typist" category; the third and fourth from the bottom (with low error-rates) were extremely careful, tending to work out commands on paper before typing them; and the fifth from the bottom had a penchant for typing random numerals, which never appeared as errors because Simper was perfectly happy to store them away. Apparently anomalous error-rates can have explanations that can improve the apparent correlation of pretest rank and error rate.

Examining the "mastery" and "perseverance" columns of Figure 19, we also see some mutual trends with pretest rank. High rankers, especially in mastery, tend to be above the median; low rankers below. Figure 20 shows similar results for Logo students. Note, however, the lack of obvious mutual trend between error rate and rank in Figure 20.

Protocols provide the following explanations. In Groups I and III: the unbridled typist returns with a friend as the fourth- and fifth-ranked students; careful planners are bottom and third from the bottom; the random-numeral typer is now caught by Logo, generating a higher rate, sixth from the bottom; and a new phenomenon: picture-printers, fifth, tenth and eleventh from the bottom, who discovered how 'PRINT' commands could be employed in procedures that "drew" their favorite things (like the "Starship Enterprise"). The latter three students made relatively fewer errors because they stagnated at this point in the curriculum. We did not intend to coerce any student to continued the curriculum, rather, we adopted a wait-and-see attitude, hoping they would eventually notice that other things, being done by other students, could also be interesting. This tack failed with one of these three students.

In Groups IV and V: paired students tended to make fewer errors because commands often were agreed upon by both partners before being typing, but pairs typically consisted of students low in the pretest ranking. One paired student, fifth from the bottom, had a partner who

("-" denotes students who worked less than 3 hours)

		Rankings Based Upon Subjective Evaluation of Performance	
Errors per Command		Mastery	Perseverance
Groups I and III (Logo data)			
.16	XXXXXXXX	1	1
.13	XXXXXXx	2	1
.28	XXXXXXXXXXXXXX	2	1
.33	XXXXXXXXXXXXXXXXx	3	2
.32	XXXXXXXXXXXXXXXX	2	2
-		-	-
.35	XXXXXXXXXXXXXXXXXx	4	5
.22	XXXXXXXXXXX	5	5
.26	XXXXXXXXXXXXX	6	5
.21	XXXXXXXXXXx	2	1
.16	XXXXXXXX	4	4
.15	XXXXXXXx	5	3
.24	XXXXXXXXXXXX	4	2
-		-	-
.26	XXXXXXXXXXXXX	2	1
.26	XXXXXXXXXXXXX	6	4
.19	XXXXXXXXXx	3	2
.28	XXXXXXXXXXXXXX	5	4
.17	XXXXXXXXx	5	3
.29	XXXXXXXXXXXXXXx	3	2
.15	XXXXXXXx	4	2
Groups IV and V			
.16	XXXXXXXX	1	1
.22	XXXXXXXXXXX	1	2
.18	XXXXXXXXX	1	1
.20	XXXXXXXXXX	1	2
.13	XXXXXXx	1	1
.21	XXXXXXXXXXx	3	3
.16	XXXXXXXX	3	4
.29	XXXXXXXXXXXXXXx	4	3
.23	XXXXXXXXXXXx	3	4
.18	XXXXXXXXX	2	2
-		-	-
.09	XXXXx	2	3
-		-	-
.20	XXXXXXXXXX	4	4
.16	XXXXXXXX	3	3
.16	XXXXXXXX	2	2

Fig. 20. Logo Students' Performance Versus Pretest Rank.

dropped out early in the experiment. She was extremely secretive about her work, which focussed upon drawing many different styles of castles via direct Logo commands -- an analogue of the 'PRINT' hang-up above.

In general, students experimented more with Logo than they did with Simper, apparently because they felt more able to express their ideas in Logo. This partially explains why the median error-rate for nongraphics, Logo students (.24) is higher than that for Simper students (.16).

If we consider our ratings of students, students' error-rates, and time needed to complete a curriculum as valid indicators of programming competence, then we can say that the preliminary test seems to be a usefully valid means for ranking programming neophytes.

6.1 Understanding the Students

This section relates most directly to our central interest: how do students learn programming and what can observations of that learning process tell us about student/tutor interactions in general. Our discussions stem primarily from protocol analysis. We consider first Simper then Logo.

Simper. We begin with the initial, untempered ideas about computers that our students brought to the course:

HELLO WHAT'S NEW?	DO YOU WANT TO PLAY JOTTO?
DO YOU LIKE SUMMER?	I AM FUNNY
THIS TYPEWRITER IS TOO SLOW	SOME DOGS ARE WHITE

```
WHAT IS 12X12?                    HOW DO YOU WORK?

TEACH ME HOW TO DO A PROGRAM      HOW DO YOU KNOW?

THERE ARE TWO MILLION FLYS IN AMERICA

DEAR JUDY, THIS COMPUTER CLASS IS A LOT OF FUN.
EVERY ONCE IN A WHILE THE COMPUTER GOES WACKEY!
```

Of course, we had encouraged the students to plumb Simper's "mind", and all the above efforts received no more than "unknown operation xxx" in reply. But a few students fortuitously struck upon operations as in:

```
001 :COMPUTERS ARE FUNNY
'are' isn't a register, use a, b, or p
001 :COMMAND YOU
'you' isn't a register, use a, b, or p
```

where 'COM' is short for the 'COMPARE' operation. This piqued their curiosity; some explored this new level of interaction ad-nauseam, but they remembered Simper's abbreviation-by-truncation for later exploitation.

Our naive programmers often had a very high opinion of computational technology. It was easy to show them that English is not yet a mode of communication between human and machine, but it took some time for the implications of this to penetrate.

At times, students' attempts at communication were tied to curriculum ideas:

```
REMARK LITERALLY                  PUT A BUG IN A

SIMPER COMMANDS ARE FAMILIAR TO COMPUTERS LIKE SIMPER

WILL YOU WRITE ME SOME SIMPER PLEASE
```

```
001 :3 4 10 ARE RELATIVE TO THE NUMBERS 15, 17, 29.
     IN WHAT WAY THOUGH?
unknown operation 3
001 :3 (THREE) IS A NUMBER AND ALL COMPUTERS LIKE YOU
     SHOULD KNOW WHAT IT MEANS!
```

Sometimes they became confused about the curriculum instructions for typing commands. The following shows some examples along with the motivating curriculum excerpt:

LINEFEED	... all you do is type LINEFEED and ...
1 TYPING 1	... and then typing 1 and ENTER ...
GO TO THE SUPERMARKET BUY EGGS AND BACON	(see Appendix 5, curriculum page 17S)
FIX PUT P 2 TO P 1 RUN	... use FIX to change ... from PUT P 2 to PUT P 1 and then use RUN and ...

In fact, some students typed Simper's prompt because it had been shown at the beginning of a line they were asked to type:

```
001 :001: ADD A 12
unknown operation 001:
```

One student tried to get a program to run by simulating Simper's runtime message:

```
007 :EXECUTING 1 TO 250
unknown operation executing
```

producing an enjoyably idiotic response. Another student, in his frustration, uncovered a bug; not in Simper, but in the Sail compiler's string run-time-routines:

```
005 :,YOU STUPID COMPUTER
'stupid' isn't a register use a, b, or p
```

The bug disguised the "," and thus the proper error: "unknown operation ,you".

Other confusions arose when students worked with both Logo and Simper (as did Group III). Logo commands cropped up in Simper protocols and vice-versa. In these cases, however, the first or second error message usually was sufficient to remind the student of which interpreter was listening to his or her typing. In a few cases, students thought they could resort to Logo commands when their Simper programs failed to produce results. By far the most common interjection of Logo was in saving programs. Apparently, learning the more complicated Logo scheme of "entries" in "files" overrode some students' knowledge of Simper's simpler filing method.

At the very least, most students initially thought that a computer could help them on a personal basis. We agree; that should, and perhaps will, someday be the case. Several discovered the '?' (or 'HELP') command which printed a general description of the Simper language. While this was never intended to be a necessary part of the course, it nonetheless was exercised frequently by a few students. Curiousity and an open desire for aid were attitudes we had hoped to exploit and certainly not to diminish. Students' were willing to experiment in trying to use Simper as an information resource to help them work out ideas from the curriculum. Unfortunately, some of their attempts were stymied by our sometimes misleading verbiage or notation.

Since work with numbers was so much a part of our students' prior schooling, it was relatively easy for them to accept that a machine (Simper) could have a good memory for numerals. But understanding that some numerals had a special meaning, other than for counting, to a machine was a more difficult concept. This was partly a problem because of the premature introduction of assembly language, thus working downward from English rather than upward from machine language. Perhaps the correct sequence would also have reduced the incidence of syntactic errors such as multiple instructions per line, making it clear that only three fields can be assembled into one location's machine-language numeral. As a result of this, we added a program which wrote over itself by reading numerals from the student. The only way this program could keep running would be if the student typed (as instructed) the numerals which were the program's very instructions. This usually clarified matters.

The orderly execution of numerals as instructions was still more abstract. The shopping list example (Appendix 5, curriculum page 17S) and the house-to-house collection (Appendix 5, curriculum page 22S) failed to motivate successor execution for some students. Programs were written with interspersed "holes", despite the obviously sequential relationship between instructions on either side of a hole. The self-destructing program mentioned above helped here as well.

Addressing remained a difficult idea for many students. One student wrote his own time-telling program, knew what had to be done to get minutes from seconds, knew something about addressing already, but typed:

```
001 :TIME A
002 :DIVIDE A 60
```

though he did not intend to divide by the content of location 60. The section on indirect addressing was very helpful to those students who still had trouble with this concept. Students who had trouble with the implicit name/value associations of the numbers-in-boxes problem on the preliminary test also had trouble with addressing in Simper.

The most pervasive problem was mastering the concept of context (or locality of information) both from the student's point of view as a user and from the point of view of instructions within his or her programs. The most common example of the former typically occurred when a student ran a program and decided that it needed modification. While it was still running, and perhaps waiting for an input (for 'CASK' or 'ASK'), he or she would type an editing command (e.g. 'LIST' or 'SCRATCH'), fully expecting it to be obeyed. Examples of the latter centered upon redundant or "clobbering" sets of instructions. For instance:

```
001 :PUT B 1                  001 :PUT B 1
002 :STORE B ONE              002 :STORE B ONE
003 :ASK A          or        003 :ASK B
004 :PUT B 1                  004 :STORE B @A
005 :STORE B ONE              005 :PUT P .-3
```

In the first program, the context within the machine is unnecessarily reset at 004 and 005; in the second, the content of location 'ONE' is continually destroyed by 'PUT P .-3'. This latter form is a common kind of bug and had already been exploited as such within the curriculum. It was apparent, however, that a much more explicit treatment of computational context was needed. Students who had the most trouble with the candy-machine problem on the pretest also had the most trouble organizing their Simper programs.

The most subtle way in which context affected the students was in the relationship among the interpreter, the assembler and the machine. Students did not fully grasp the distinction between editing commands and assembler/machine instructions. Sometimes they attempted to abbreviate the former (e.g. "SCR" for 'SCRATCH') and expect the latter to be obeyed at once. Again we modified the curriculum in an attempt to clarify these issues, some of which were founded upon confusing editing time with execution time.

Some of the "holey" programming can be traced to Group III students who learned to use Logo line-numbers in canonically sparse (10-20-30...) sequence and hoped the same editing advantages would accrue in Simper.

Toward the end of the curriculum, procedures and their calling sequences provided examples of how programs could be structured by writing functionally related subunits. In this case, holes were ok. Success here demanded that the student had mastered the concepts of addressing and program control. Failures to structure these programs correctly were of two forms: failure to define a proper calling sequence, and misplacement of the calling sequence in the flow of the program. Some inputs to procedures, particularly the return address, were overlooked; once the call itself was incorporated as part of the procedure body.

Because no students had time to do significant work on the final part of the curriculum dealing with stacks and recursive procedures, we lack some potentially interesting data. The notion of context could perhaps be motivated very well here. However, our sequential pass through the curriculum has failed to mention that we have been relying increasingly upon data from the more able, typically older, students. The remaining students simply did not proceed as far. How this has colored our observations we can't say, but we can say that students who had trouble seeing any valuable return for their relatively large programming efforts (in comparison with Logo) tended not to proceed with the curriculum and felt that they would "never understand Simper". And these students were usually, but not always, less than the most able.

A few miscellaneous comments remain. Some students were active in exploiting features of the Simper interpreter -- for instance, the truncation of operation names (e.g. 'STOP' for 'STORE' and 'LOAN' for 'LOAD'). One student occasionally seemed to harass the machine by repeatedly saving a program on a file that already existed just so he could respond "no" to Simper's warning: "a program called xxx already exists! ok to destroy it?". The importance of clear, relevant error messages also became apparent (see Section 5 for examples). An example of how misreading one word can dangerously alter the meaning of a message:

```
010 :SAVE
what do you want to name your program? YES
ok, yes is saved
```

illustrates the care that must be applied to apparently trivial aspects of an interpreter. In line with our earlier comments about contextual errors, we should mention that the above question and the students together produced a large number of saved programs called 'SCRATCH'.

Logo. After showing students how to log in to the system, the curriculum encouraged them to type any line they wished to the Logo interpreter. This was for the purpose of demonstrating Logo's understanding of English as well as giving us some understanding of what the students expected the computer to be: numerical calculator, omniscient authority, or game player:

```
COMPUTERS ARE DUMB                THIS IS GOING TO BE VERY FUN
computers needs a meaning         this needs a meaning.
COMPUTERS ARE ILLOGICAL           IT MEANS IT WILL BE ENJOYABLE

HOW MANY QUESTIONS CAN YOU ANSWER?

HOW LONG HAVE YOU BEEN IN SERVICE?
how needs a meaning
YES                               MY DOG IS BLACK
yes needs a meaning.
AFFIRMATIVE                       THE SUNSET IS BEAUTIFUL
affirmative needs a meaning.
YES MEANS AGREED, CORRECT         PLAY CHESS
yes needs a meaning.              play needs a meaning.
I JUST GAVE YOU A MEANING         PLAY MEANS TO DO SOMETHING
i needs a meaning.
I MEANS #176                      GET GOLF
i needs a meaning.                something missing for get.
I GIVE UP                         GET GAME
                                  something missing for get.
HOW MANY WORDS DO YOU KNOW?       GET PLAY
                                  something missing for get.
WHY ARE YOU A COMPUTER?           YOU ARE A STUPID COMPUTER

WHERE IS GERMANY?
you are not using the train
YES I AM
```

After learning about literals and several commands, Groups I and III began Part 4, while the graphics groups spent several days drawing pictures with the direct commands learned in Part 3 -- two examples were a drum set and a cube with legs. While we did not want to stifle individual expression and force students to go through the curriculum at the same rate, we also did not want them to grow addicted to immediate results and frustrated by the lack of more powerful tools (i.e. procedures) through which they could edit, store and reproduce their pictures. These factors may be responsible for several of our early dropouts. Had procedures been introduced at the beginning, a student would have had a framework within which to execute direct commands and then add them to his/her procedure via simple editing commands.

Students in Part 4 sometimes forgot to quote literals, either as names or values, 'MAKE "ALPHABET" ABCDEFGHIJKLMNOPQRSTUVXWYZ'; they reversed name and value positions, 'MAKE SUM OF 5 AND 9 ANSWER'; or they attempted linked assignment through one command (not an unreasonable expectation), e.g. 'MAKE "SNOOPY" "CHARLIE BROWN" "LINUS"' (where the curriculum intended 'MAKE "SNOOPY" "CHARLIE BROWN"' and 'MAKE "CHARLIE BROWN" "LINUS"').

Some initial confusion about Logo's colon notation (i.e. 'THING OF "SNOOPY"' could be written alternatively as ':SNOOPY:') resulted from the poor quality of our photocopied lineprinter listings: some students mistook ";" or "!" for ":". Others tried expressions such as '::SNOOPY::' to mean 'THING OF THING OF "SNOOPY"', but Logo (inconsistently) does not permit the nesting of colons. Logo allows numbers to be names also but this later led to some unfortunate confusions between literals and names, and actual and formal parameters.

Armed with the procedure examples from Part 5 and the 'PRINT' command, many students chose to make posters and print endless statements.

```
TO TOM
10 PRINT "IF TOM WAS NOT GREAT I WOULD STOP WRITING"
20 TOM
END
```

Pictures of various Star Trek ships appeared ('KLINGON.BATTLE.CRUISER', 'ENTERPRISE', and 'GALILEO') as well as interpretations of the human anatomy. Students wrote procedures for each letter of the alphabet and other procedures to type messages one letter at a time. For example:

```
TO HELLO
10 HHH
20 EEE
30 LLL
40 LLL
50 OOO
END
```

Still the concept of a literal string was not firm in student's minds: strings often appeared unquoted or partially quoted when involving the special character "#" (used by Logo to provide additional "real" blanks since all but one blank between words is deleted when sentences are stored). Students may have believed that # was a command to type blanks and therefore (correctly) did not quote it. After Part 6 (and with the aid of the first page of Part 9), the students could specify messages as Logo strings, e.g. 'POSTER "HELLO"', 'POSTER "TELETYPES HAVE PORNOGRAPHIC MEMORY BANKS"' and 'POSTER "THIS SIGN HAS NO MEANING IT HAS NO INPUT AND GIVES NO OUTPUT"'. Despite high enthusiasm, many students abandoned their own tool-building efforts when they discovered that a poster program (Snoopy carrying a nicely formatted sign) already existed.

Graphics students wrote procedures of analogous complexity (examples below) but producing results on the display screen often did not provide the same reinforcement as producing hardcopy to take home. The plotter was late in arriving, and our attempts at photographs were poor since our borrowed camera lacked a proper hood attachment.

```
TO DANCE            TO WHEE              TO CARDS
10 POKE             10 FRONT RANDOM      10 SQUARE
20 UNPOKE           20 RIGHT RANDOM      20 RIGHT 10
30 RIGHT 90         30 WHEE              30 CARDS
40 DANCE            END                  END
END
```

Students sometimes forgot to provide input names in the 'DOUBLE' procedure or spelled them differently from occurrences in the procedure body; they put colons around the numeral 2 ('OUTPUT PRODUCT :NUMBER: :2:'), forgot the command for multiplying (should the name reflect the operation ('MULTIPLY') rather than the result ('PRODUCT')?), or squared the number instead of doubling it ('OUTPUT PRODUCT :NUMBER: :NUMBER:'). Since Logo accepts noise words such as 'OF' and 'AND', many students expected to be able to use "BY" in the division command used in their 'UNDOUBLE' procedure. This led us to question the use of Logo noise words at all, and suggested that students should be able to add their own sets of noise words. Some examples of these problems follow.

```
TO UNDOUBLE (missing input)           OUTPUT QUO :NUMBER: BY 2

TO UNDOUBLE IS TO TAKE HALF           OUTPUT QUO :NUMBER: :2:

TO UNDOUBLE :THING OF :NUMBER:        OUTPUT QUO :NUBER: :NUMBER:

UNDOUBLE MEANS TO DIVID               OUTPUT DIV 2 :NUMBER:

PRINT DIVIDE :NUMBER: BY 2            OUTPUT QUO OF :NUMBER:
                                                AND :NUMBER: BY 2
PRINT DIVISION :NUMBER: :NUMBER:

PRINT QUOTIENT :NUMBER: DIVIDED BY 2
```

The next problem in Part 6 was a functional relation taken from the preliminary test; it was also used in the Simper curriculum:

> "What function (rule) using only simple arithmetic, can you find which changes
>
> | each of these numbers | into each of these numbers |
> |---|---|
> | 3 | 15 |
> | 4 | 17 |
> | 10 | 29 |
>
> Write a procedure that uses this rule. (If you don't know the rule or how to start the procedure, ask a tutor)."

We hoped that students would use their 'DOUBLE' procedure in the solution:

```
TO RULE :NUMBER:
10 OUTPUT SUM 9 AND DOUBLE :NUMBER:
END
```

But those not using 'DOUBLE' often became entangled in the mysteries of nested expressions, noise words and syntax in trying to produce: 'OUTPUT SUM :NUMBER: AND SUM OF :NUMBER: AND 9'. Other examples are:

```
OUTPUT SUM :NUMBER: :NUMBER: 9          (missing a SUM)

OUTPUT SUM :NUMBER: :NUMBER: SUM OF 9   (SUM in wrong place)

SUM OF 9 TO THE PRODUCT OF :NUM: BY 2

TO CORRESPOND 3 TO 15, 4 TO 17,         (an interesting but
                         AND 10 TO 29    illegal title)

TO ADD :NUMBER:
10 OUTPUT SUM DOUBLE ADD 9              (unbounded recursion)

MULTIPLY :NUM: BY 2
ADD 9

MAKE PROD :NUMBER: AND 2 ANSWER
OUTPUT SUM OF ANSWER AND 9
```

In the last two examples, students appeared to understand the "rule" but tried writing the expression on two sequential command lines, forgot the names of the 'PRODUCT' and 'SUM' operations, did not quote names or reversed the name and value inputs to 'MAKE'.

In drawing complex pictures, students were encouraged to decompose these into more basic geometric shapes. They were helped in writing programs to draw shapes of any size. In three of the following 'RECTANGLE' examples (all from different students), errors indicate that the students may be trying to give values to the procedure inputs at define time; in addition, there is a curious lack of the commands ('FRONT') to do the actual line drawing.

```
TO REC :LENGTH: :WIDTH:         TO RECT :LENGTH: :WIDTH:
10 PD                           10 20
20 :LENGTH:                     20 50
30 RIGHT 90                     END
40 :WIDTH:
END                             TO RECT :LEN: :WID:
                                10 OUTPUT RECT 6 3
TO RECT :LEN: :WID: :200: :50:  END
```

The first problem in Part 7 was analogous to 'DOUBLE' for strings:

> "Write a procedure called AGAIN that doubles its input word (its input is a word), and outputs the resulting word.

```
When you type this              you should get
------------------              --------------
P AGAIN "DOG"                   DOGDOG
P AGAIN AGAIN "ALDO"            ALDOALDOALDOALDO
P AGAIN W "BL" "ACK"            BLACKBLACK
P AGAIN :EMPTY:
P AGAIN 12345                   1234512345      "
```

Errors in solving this problem (examples follow below) and other problems from this section involve coordinating procedure inputs, the correct functional operations, and the 'OUTPUT' command. Students forgot to put input names in the title, used literals in place of names, or used names different from those named in the title. For these latter, Logo happily supplies the default value ':EMPTY:' rather than complaining about an undefined variable. Inconsistently, undefined procedures are not defaulted to "no-ops", and a name and its default instantiation do not appear when the student requests a list of all the names in the workspace. Students sometimes substituted 'PRODUCT' for 'WORD'; the noise word 'AND' appears in several contexts suspiciously like an infix concatenation operator -- another reason why default noise words should be eliminated, certainly those which have a strong, clear meaning in natural language.

```
TO AGAIN :W:
10 OUTPUT WORD :W: :W:           (a solution)
END

P "INPUT" @ W "INPUT"            P WORD AND WORD

P "WORD" PLUS "WORD"             OUTPUT WORD OF "WORD" AND "WORD"

OUTPUT PRODUCE :LETTERS: 2       :WORD: REPEAT

OUTPUT :WORD: AND :WORD:         :W: WORD :W:

OUTPUT WORD :DOG: :DOG:

OUTPUT INPUT :WORD: AND :WORD: AGAIN
```

Students wrote procedures to return the second and third letters of a word. These were intended as building blocks for a procedure called 'SWITCH13' which would exchange the first and third letters of a word. Students often failed to break the problem into manageable parts and thereby notice that some of the components had been solved previously.

We were looking for solutions of the form 'OUTPUT WORD WORD WORD THIRD :W: SECOND :W: FIRST :W: BF BF BF :W:' ('BF' is the abbreviation for 'BUTFIRST', 'F' is the abbreviation for 'FIRST'). Several students mentioned the input only once (i.e. 'OUTPUT W W W F BF BF F BF F BF BF BF :W:') -- they may have believed that :W: was automatically distributed and that they need only mention it once. The following is typical of the half Logo, half English procedures which occasionally appeared.

```
TO SWITCH13
10 THIRD :INPUT:
20 FIRST :INPUT:
30 PUT THIRD FIRST AND FIRST THIRD
END
```

Students in the teletype groups were asked to write a recursive procedure to type dashed lines where the "dash" could be any character. Simper students worked on a similar problem. The graphics students worked on a 'DASH' procedure with one input for the visible part and the other input specifying the gap length. Example attempts include:

```
TO DASH                  TO DASH :DIS: :TANCE:
10 RUN                   10 PENDOWN
15 FRONT 100             20 FRONT :DIS:
20 PENUP                 30 PENUP
25 FRONT 10              40 FRONT :TANCE:
30 PENDOWN               50 DASH :DIS: :TANCE:
35 LINE 15 DASH          60 DASH :DIS: :TANCE:
END                      END
```

Since we did not teach about Logo's 'GOTOLINE' command, the example on the left must contain influences ('RUN' and 'LINE 15 DASH') from Basic or Simper. The 'DASH' procedure on the right does draw dashed lines, but the extra line 60 indicates that there is some doubt in the student's mind about how recursion works -- perhaps she expected only three dashes to be drawn (e.g. by lines 10-40, 50, 60).

Teletype students produced a diagonal dash program and tried a recursive procedure with a changing input to do ripple printing, which is similar to an earlier problem, i.e. move the first letter to the end of the word, repeatedly: BANANAS, ANANASB, NANASBA, etc. Graphics students wrote a procedure to bend lines (i.e. make polygons) with two inputs: the distance the turtle moves and the angle to turn each time, and then modified it ('BD' below) to make spirals (they later made 'BD' stop after some number of iterations).

```
TO SQUEER :SZ: :TU:              TO BD :L: :A: :I:
10 SQUARE :SZ:                   10 FRONT :L:
20 PENUP                         20 RIGHT :A:
30 SQUEER DIFF :SZ: :TU: :TU:    30 BD :L: SUM :A: :I: :I:
END                              END
```

The 'SQUEER' procedure makes patterns of nested squares. Students enjoyed experimenting with different number inputs to these procedures. One frequent error was forgetting to specify all of the inputs in a direct command or recursive call, especially when that input does not change: for example omitting the last :I: in line 30 of 'BD'. One student defined the following unusual construct:

```
TO STEVE :BD 17 16 48:
10 :BD 17 16 48:
END
```

She then typed 'STEVE BD 17 16 48', which incidentally works because in attempting to bind STEVE's input, Logo runs BD and waits for a value, which never comes. We suspect that the student did not realize this;

trying 'STEVE' with a different call to 'BD', with 'STEVE' and 'BD' traced, would have helped correct this mistake.

For an 'EVENP' procedure which returns '"TRUE"' if its number input is even, and '"FALSE"' if it is odd, students had trouble with distinguishing 'OUTPUT' and 'PRINT' and expressing numerical tests (infix expressions might be more natural). 'OUTPUT', from the students' point of view, was apparently not the best choice of words, hence we added 'RETURN'. One might also consider words like 'REPLY', which tend to better describe the message-sending/receiving activity going on during Logo's evaluation.

```
TO EVENP :NUMBER:                             (one solution:
10 TEST ZEROP REMAINDER :NUMBER: 2            these three lines could be
20 IFTRUE OUTPUT "TRUE"                       replaced by OUTPUT ZEROP
30 IFFALSE OUTPUT "FALSE"                     REMAINDER :NUMBER: 2)
END

OUTPUT "TRUE" IFTRUE DIVIDEND OF :NUMBER: 2 LEAVES NO REMAINDER

IFEVEN P TRUE                                 P FALSE IF NOT DIVISABLE BY 2
IFUNEVEN P FALSE                              P TRUE IF DIVISABLE BY 2

OUTPUT "TRUE" IFTRUE EVEN
OUTPUT "FALSE" IFFALSE ODD

TO HORZDASH :PAPER:
10 TEST ZEROP :PAPER:
20 TYPE "-#"                       (result of the test is ignored)
30 HORZDASH DIFF :PAPER: 1
END
```

Students wrote procedures to type dashed lines and make their teletypes sound like ringing telephones; graphics students made the turtle dance by poking ('TURTLEPOKA') its head out on even degree turns, and pulling it in on odd degree turns.

Binary "life" (see Appendix 4) uses a Logo string of 0's and 1's as a colony of reproducing creatures on the planet Binar, with each new generation appended to the colony based on the oldest (leading) generation which then dies. The colony becomes extinct if it ever fell below a certain critical size. This project was quite popular with students. They tried to predict which initial states would result in expansion, steady state or extinction. One student also experimented with changing the rules.

Parts 9 and 10 contained numerous examples of projects combining most of the earlier concepts, but, unfortunately, few students started these sections; reasons include vacations, involvement with two languages and curricula (Groups I, II, and III), and other projects.

Students embarked on several projects of their own choosing. A "Madlibs" program typed "story skeletons" with certain nouns, verbs, adjectives and adverbs supplied by the program user, often with hilarious results. The use of one procedure input per missing word worked well for small numbers of words but needed a more general scheme for prompting and obtaining values. In the days of pre-animation graphics, a launch procedure gave the illusion of a rocket trajectory by drawing and erasing a gradually shrinking, rising and tilting rocket. One ambitious student drew a pool table with pockets, cue stick, rack and balls. He then programmed a movie sequence for the break and several shots via 'ZAP'-ping and redrawing appropriate parts of the scene. One graphics student abandoned his drawing activities for several days in order to make an elaborate poster saying "SIMPER IS

FULL OF SUDS". Another student produced what we thought was a novel approach to drawing a circle with a turtle:

```
TO CIRC :RAD:
10 FRONT SUM :RAD: 1
20 PENDOWN
30 BACK 1
40 PENUP
50 BACK :RAD:
60 RIGHT 1
70 CIRC :RAD:
END
```

In closing, we mention several examples of student behavior. One girl usually began her session by typing 'MAKE "GAIL OLDS" "THE GIRL THAT IS TYPING ON ME"'. Several of the graphics students spent the remainder of the morning on teletypes after their regular hour was done. Some students logged in on several terminals so that they could make posters while programming. General reactions to errors ranged from logging out (using 'GOODBYE' or their own logout or "selfdestruct" procedures), to shifting tasks (trying familiar procedures they had written earlier), random typing (many "carriage returns" on graphics terminals eventually scrolled the text area, thereby erasing the error message), asking tutors for help, or trying to respond in an intelligent (although English) way.

```
MAN                                  STICK IT IN YOUR EAR
man needs a meaning.                 stick needs a meaning.
MAN DOESN'T NEED A MEANING, YOU DO   TO INSERT

SEE
you are not on an imlac
WHY NOT?                             WHEN I FINISH TYPING THIS LINE
                                     LINE YOU WILL REPEAT IT AFTER
illegal mem alloc trap..from 13125   YOU TRANSLATE IT INTO YOUR
ILLEGAL MEN IN ALLOC TRAP            MEMORY BANK
```

6.2 Evaluation of Simper and Logo

As a result of the experiment, we have various modifications that we have made or would like to make to the languages and devices used by our students. We will discuss these along with general commments about their pedagogical usefulness.

Simper. First targets for change were obvious bugs and inconsistencies in command evaluation and assembly. For example, 'SCRATCH' was modified to accept the general form for an address-range specification (e.g. 'SCRATCH 6:8' has the obvious effect). 'SAVE' and 'GET' now accept the name of the file as an input (e.g. 'SAVE GLOP'), resorting to the dialog mentioned earlier only when an input is lacking. A more subtle change was made to 'SLIDE'. One student was frustrated when his memory space was effectively exhausted even though numerous holes existed between program segments. A forward 'SLIDE' (e.g. 'SLIDE 100:200') now recursively squeezes out such holes to make formerly impossible relocations possible. The user is informed of which holes disappear.

In the interest of making the name fit the action, 'FIX' was replaced by 'EDIT'. This was also done to reduce the language burden of learning both Logo and Simper.

New operations and a new command were added. 'LEXOR' gives a decimal version of "exclusive or" (Table II), 'ERROR' tests for arithmetic overflows, 'IOT' communicates with the Graphics program and the plotter, and 'NEWS' gets the system time schedule and any new information about Simper (or Logo). 'DIVIDE' was modified to set a flag, detectable by 'ERROR', on division by zero, instead of the previous and unusual skip-if-successful convention.

The structure of the Simper machine itself was modified. Five-hundred memory cells and four registers (i.e. A, B, C and P) were made standard (with upper limits as shown in Figure 2). This was motivated by students suggesting projects for which 250 memory cells would be insufficient. The additional register was added to make procedure calls more convenient, especially via a student-programmed stack. The changes were achieved by a generalized restructuring of the interpreter.

Recommendations. Changes are relatively easy to make in Simper because it is written in a high-level language. An important improvement would be the simulation of a micro-coded machine with interrupt handling, so that students could be exposed to some aspects of modern machines. Simulated devices other than the turtle (e.g. a disc) could also be pedagogically beneficial. However, too many "features" can be detrimental. Since one of the most valuable computational ideas is that problem solutions can be broken logically into parts that are in turn realized by certain basic and sufficient abilities of some machine, the abilities chosen should not be too powerful.

Perhaps the most beneficial results would be achieved by making the interpreter smarter and more congenial in terms of its responses to naive programmers. A first step would be a structured treatment of the '?' or 'HELP' command. Successive applications of this command in, say, an address field would obtain successively more detailed help about address fields. In this respect, the interpreter would be more knowledgeable about itself. More general (and more difficult) powers, such as the ability to evaluate programs, would be of obvious value in counselling students.

Except for a few run-time bugs introduced by the Sail compiler, the Simper interpreter proved remarkably durable under student use. No student ever lost his active memory or a saved program as a result of an interpreter fault.

Logo. In our present version of Logo, changes such as editing error messages, adding new commands, and substantial changes to the parsing and naming schemes range from easy to painfully difficult. Had more resources been devoted to this project, we would have designed and implemented our own Logo interpreter in Sail. Integrating this with existing Sail software (i.e. train, graphics, animation) would provide greater accessibility to data objects such as snapshots and avoid the multiple-process structure of Tenex and the associated time penalties incurred in using graphics and animation. In addition, it should be easier to experiment with old and new features (e.g. parsing, filing and program analyzing) and to use this as a model for implementing subsets of Logo in other languages (e.g. Basic or Fortran) for users of other, smaller machines. Another option we considered was to modify an as yet unavailable interpreter (Manis, 1973) written in Bcpl, which is generally machine independent.

Recommendations. If ':X:' is to be analogous to 'VALUE "X"', then nesting of colons should be allowed. Additionally, a different symbol should be used instead of colon to delimit place holders in procedure titles, or a different synonym for 'VALUE' could be chosen (e.g. "@"). Numerals should be disallowed as names. More fundamentally, we suggest that atom names and procedure names use the same dictionary and notation (e.g. 'A' could either stand for 'VALUE "A"' or call procedure 'A', as in Algol 60). Pedagogically speaking, any distinctions of program from data should be defined by the student and not be automatic.

Command Evaluation. When a student types the 'MAKE' command but omits one or both inputs, Logo prompts for the missing inputs (after typing "NAME" or "VALUE" as appropriate) rather than yield an error message as is the case with all other commands. Control over evaluation and error handling must be more generally accessible in order to be a useful rather than an inconsistent feature. Commands for editing, erasing, listing and filing currently quote rather than evaluate their inputs (i.e. 'EDIT ROCKET' instead of 'EDIT "ROCKET"' thus disallowing 'EDIT :R:' where 'VALUE "R"' is "ROCKET") A consistent, flexible scheme (assuming names and procedures share the same name table as suggested above) would allow only 'EDIT "ROCKET"' and 'EDIT R'. 'EDIT ROCKET' could also be allowed if the user could make his own procedure definitions that quote or evaluate inputs at will -- all in the interest of consistency, which is very important to naive programmers. A further simplification would result if one operation (e.g. 'DEFINE' or 'HOWTO') performed the functions of both 'EDIT' and 'TO', since the only difference is the pre-existence of, or lack of, a definition.

Noise words should be eliminated unless they are under user control. Logo should emulate Lisp in returning values for all commands and perhaps printing these values at the top level rather than giving the message "THERE IS NO COMMAND FOR..." when a student forgets to precede a function call with a receiver for its reply. A user-controlled toggle for auto-value-printing would be a useful debugging aid. This would make 'STOP' and 'DONE' equivalent to 'RETURN ""'.

Logo should allow multiple commands on a line, even though these would be more difficult to edit and might introduce ambiguity. Then the semicolon comment-toggle would truly make sense.

Error Messages and Primitive Names. Synonyms for commands (e.g. 'REPLY' or 'RETURN' for 'OUTPUT', and 'DONE' for 'STOP') have been added to clarify certain concepts. Error messages such as "OUTPUT CAN'T BE USED AS AN INPUT IT DOES NOT OUTPUT" have been edited. Misleading messages such as "OUTPUT CAN ONLY BE USED IN A PROCEDURE" require additional logic to determine context: in response to the situation of typing 'OUTPUT' as a direct command during procedure editing, the message should be that 'OUTPUT' cannot be a direct command and should be preceded by a line number. Error messages should not end with a "?" unless the interpreter is able to engage the student in a helpful dialog.

Editing and Filing. One common desire was to change a line in a procedure with one rather than two commands -- commands such as 20 and 'EDL 20' typed at Logo's top level resulted in the messages "LINE 20 OF WHAT PROCEDURE?" and "EDIT WHAT? YOU ARE NOT DEFINING ANYTHING" which may have misled students into trying the following commands:

```
EDIT LINE 10 OF UNDOUBLE          EDIT LINE 10 IN TRI2

ERASE LINE 6 IN RECTANGLE         P LINE 15 IN STOP

TO 35 OF RECTANGLE                TO @35 OF RECTANGLE

TO "35" OF RECTANGLE
```

Since we view line numbers as an editing convenience for non-display devices rather than as necessary labels for controlling procedure evaluation, continued work with graphics terminals should include experiments with different editors.

Students often included extra words (some which Logo used in messages), noise words, or expressions in commands such as 'EDIT', 'ERASE', and 'LIST', which do not obey the general Logo evaluation scheme; hence, error messages were often puzzling.

```
EDIT TO EVENP                     EDIT :XI:
you can't edit that.              you can't edit that

ERASE :XI:                        ERASE TO SQUARE
erase what?                       erase what?

END                               LIST ALL FILES
again defined                     list all what?
UNDEFINE AGAIN
undefine needs a meaning.         LIST NAMES
                                  something missing for list.

LC OF FILE OF MARTA               LIST ALL THAT WAS DONE TODAY
of can't be a file name           list all what?

GET GAIL FILE                     GET FILE PC136 VOWELP
file can't be a file name.        file can't be a file name.
```

As a convenience, it might be helpful to allow some default applications of operations like 'LIST'. For instance, when 'LIST', 'EDIT', 'ERASE' or 'EDIT LINE xx' is typed with no input, the default input would be the name of the last procedure 'END'ed or the last procedure in which an execution error occurred.

The distinction between what is in Logo's immediate memory (workspace) and what is on secondary storage (file entries) is confusing even to adults. By saving an entire workspace on an "entry", it is fairly easy to 'GET' everything back at a later time. But since the workspace could contain the appended results of several 'GET's from other entries (from other people's files too), there is often unnecessary duplication in 'SAVE's. One should have the ability to save partial workspaces (groups of procedures) on entries.

```
SAVE GAIL TRIANGLE       (this replicated Gail's workspace
SAVE GAIL RECTANGLE      in three separate file entries)
SAVE GAIL REPEAT

SAVE LIZ D AND UD AND SQUARE    (Liz wanted to save individual
                                procedures on separate entries)
```

We found examples of student typing, some almost verbatim from the curriculum, which we might expect a reasonable computer-based tutor to be able to handle. The naive approach of merely automating a programming curriculum (such as ours) by typing text at the student will accomplish little in dealing with such questions. We believe that the language interpreter should "know" something about what concepts and problems the curriculum is presenting and the intents of procedures the student is writing.

```
HOW MANY INPUTS DOES "MAKE" HAVE?

IS REQUEST A LITERAL?
literal needs a meaning.
NO IT DOESN'T
```

HOW MANY INPUTS DOES PRINT HAVE

IS "GEORGE" A WORD?

The ability to answer these questions is easily given to Logo because the subject terminology (perhaps excepting "literal") is Logo's.

Since Logo already checks procedure lines for matching quotes and colons at the time they are typed, it would seem advantageous to report other kinds of syntax errors at "define-time" rather than at "run-time". For example, erroneous number of inputs for primitive commands and user procedures, and undeclared procedures or names (not defined globally or in the procedure's title) could be reported after every line typed, before taking the student out of "editing mode", or upon request. The student can act on these suggestions and make further editing changes, execute the partially defined procedure while still in editing mode, or exit to work on something else. Although this would be of little help in detecting semantic errors, it could serve to minimize the amount of time students spend in discovering and correcting syntax errors one at a time.

6.3 Implications for Language and Curriculum Design

Reports of tutors about student involvement in different parts of the curriculum and their own projects, real or planned, led us to make curriculum changes involving the order of the concepts presented and techniques for explaining certain concepts. For example, names were viewed as belonging between literals and procedures in complexity, and as prerequisites for procedure inputs and list-like data structures.

For Simper, most changes centered upon better motivations for: context, sequential execution, addressing and assembly language. The machine's language of numerals would be taught before assembler syntax so that students would grasp the latter's reason for existence as well as its structure. The fact that different languages are appropriate for different interactions with Simper would be exploited in teaching about computational context. The intercommunication of instructions (e.g. via the registers) within programs would also be treated in terms of context.

We found that students were not particularly motivated by Logo-Part 4 because, for all their efforts, only a few strings appeared on their terminals. Introducing the turtle commands in the context of the first procedure example would have allowed students to start editing and saving their initial pictures rather than using the less enduring direct commands. As a result, names should be introduced first when procedures need them as inputs (formerly Part 6), and procedures should be introduced immediately after literals.

If testing had been introduced earlier than Part 8, where its chief use was to provide stopping rules for recursive procedures, students could have embarked earlier on their own projects, e.g. games like Blackjack and guessing numbers. This also has the advantage of not compounding testing with already difficult concepts behind recursive procedures with changing inputs.

The graphics curriculum must provide a more clear-cut case for the advantages of graphics. By replacing the numerous teletype examples by problems using graphics and animation, while maintaining a parallel ordering of concepts, results of comparing teletype and graphics curriculum can be more meaningful; in addition, two separate but parallel problem domains can provide for interesting testing of transfer for students from each curriculum. The curriculum can be enhanced by a computer-based tutor's use of graphics in presenting examples (one use is discussed below) and providing editing and debugging facilities.

The curriculum format of path pointers, questions, problems and things to try was generally well-received by students. Certain connecting ideas or processes such as how expressions are evaluated and how procedure evaluation proceeds are difficult to sequence on paper, and the flowchart-like diagrams with boxes and arrows we tried were not particularly effective. The younger children had especial difficulty with these artifices, for the same reasons they had trouble with the candy-machine problem on the preliminary test. Good yet static representations of essentially dynamic processes are hard to come by. The "brothers" with knowledge clouds did test understanding when some of their states were left blank, but were of little help in mapping this understanding into a Logo procedure. One possibility is that Logo could graphically simulate some of its own internal workings. As an example of this idea, a film animating the evaluation of a Logo procedure has been done by Ron Baecker and students at the University of Toronto (Baecker, 1974).

7 Final Comments

Having modified both curricula and interpreters and gained confidence on the extent to which the preliminary tests predict success in the original curriculum, we proceeded to repeat the experiment with smaller, better controlled groups of students and more uniform tutoring. Analysis of this second effort, and a more thorough study of Groups I, II and III of the experiment we have been discussing, will appear in a separate report (Cannara, 1975).

Appendices

The appendices included here pertain to the summer experiment of 1973. The states of Logo, Simper and the curricula are reflected here as they were during that experiment, unless noted otherwise. Some Logo operations fit into none of the following appendices, so they are included here.

Miscellaneous IMSSS Logo Primitives

Any abbreviations are included beneath the full operation names, the number of inputs (arguments) required is noted in parentheses with the description of each operation, and "$" indicates operations which were implemented after, and partly as a result of, the experiment discussed in this report.

ASCII	(1-input operation) the input is the octal ASCII code of the character to type, e.g. ASCII 10 types a backspace
ASKCHAR ASKC	(1-input operation) return a character from the terminal (the input is the number of seconds to wait before returning :EMPTY: if nothing is typed, see ASK)
ASSIGN	$ equivalent to MAKE (not implemented)
BLANK	(0-input operation) equivalent to TYPE :BLANK:
BREAK	(0-input operation) interrupt program execution as if CTRL-G had been typed (see GO, CANCEL)
DEFINE	$ equivalent to TO (not implemented)
DONE	$ equivalent to STOP
HOWTO	$ equivalent to TO (not implemented, but TO could be HOWTO's abbreviation)
RAND	(2-input operation) return a random integer from the range: (first input, second input)

REPLY	$ equivalent to OUTPUT (not implemented)
REQUESTCHAR RQC	(0-input operation) return a character from the terminal (see REQUEST)
RETURN	$ equivalent to OUTPUT
SAMEP	$ (2-input operation) equivalent to IS
SAY	(1-input operation) audio system speaks a word or sentence, spelling any words for which it has no unitary, prerecorded sounds
VALUE	$ (1-input operation) equivalent to THING
WAITM	(1-input operation) the input is the number of milliseconds to wait (not very accurate when the Tenex system is busy, see WAIT)

Appendix 1: Graphics

In this appendix, we describe in some detail the two types of graphic devices available to Logo users: the TEC(R) and IMLAC(R) displays. Tables of the relevant Logo operations are included. IMSSS Logo is aware of the various types of terminals available to users and so can correctly execute some operations in more than one way, automatically producing an effect appropriate to the device at which the user happens to be (e.g. 'LEFT' works for the TEC, the IMLAC and other devices, Appendix 2). In the following tables, any abbreviations are included beneath the full operation names, the number of inputs (arguments) required is noted in parentheses with the description of each operation, and "$" indicates operations which were implemented after, and partly as a result of, the experiment discussed in this report.

1.1 TEC$^{(R)}$

The TEC is a raster-scan (video) display whose screen is refreshed from a shift-register memory. It cannot be made to draw lines, but it possesses extensive abilities for editing the text appearing on its screen. Under program control, these abilities are accessed by sending certain ASCII characters (some are "control" characters, some are lower-case) to the device. The user's typing is restricted to upper-case. The abilities thus available to a Logo user are described in the following table, after which a sample Logo program that simulates an elevator is included.

IMSSS Logo TEC$^{(R)}$ Primitives

BLINKOFF	(0-input operation) terminate blinking region at cursor
BLINKON	(0-input operation) start screen blinking to right of and below cursor
BOX	(0-input operation) type a "box" character
CLEAR CS	(0-input operation) clear the screen and then HOME
DELETECHAR DC	(0-input operation) erase character at cursor position and move the rest of line left one character
DELETELINE DL	(0-input operation) erase line cursor is on and move lower lines up one line; move cursor to beginning of the line
DOWN	(0-input operation) move the cursor down one line (wraparound from bottom to top of the same column)
ERASEDOWN EEOP	(0-input operation) erase the screen to right of and below cursor
ERASERIGHT EEOL	(0-input operation) erase rest of line to right of the cursor
HOME	(0-input operation) move the cursor to the upper left corner of the screen (0,0)

INSERTCHAR IC	(0-input operation) insert a blank character at cursor position and move rest of line right one character
INSERTLINE IL	(0-input operation) insert a blank line at cursor position and move lower lines down one line (last line is lost) move cursor to the beginning of the line
LEFT	(0-input operation) move the cursor left one character (wraparound from left edge to right edge of row above, and from top left corner to bottom right corner)
MOVEXY	(2-input operation) move the cursor to absolute screen position (the column (first input) is between 0 (left) and 79 (right), the row (second input) is between 0 (top) and 23 (bottom))
RIGHT	(0-input operation) move the cursor right one character (wraparound from right edge to left edge of row below, and from bottom right corner to top left corner)
UP	(0-input operation) move the cursor up one line (wraparound from top edge to bottom of same column)

The following is a listing of a Logo elevator simulator that uses some of the operations above to "draw" and move a 10-story elevator on the TEC screen. 'START' initiates the simulation.

```
TO START                   ; initializes the simulation
10 CLEAR
20 MAKE "CAPACITY" 10      ; elevator's size
30 MAKE "FLOOR" 10         ; building's height
40 MAKE "CENTER" 36
50 MAKE "BOTTOM" 22
60 PEOPLE                  ; put people on the floors
70 SKIP
80 MAKE "QUICK" 250
90 MAKE "SLOW" 900
100 MAKE "FLOOR" RAND 1 10
110 MAKE "ELEVATOR" RAND 0 :CAPACITY:
120 RUN
END

TO RUN
10 TEST ZEROP SUM THING WORD "FLOOR" :FLOOR: :ELEVATOR:
20 IFTRUE SHOW :QUICK: FLOORPOSITION
30 BELL
40 IFFALSE SHOW :SLOW: FLOORPOSITION
50 RIGHT
60 UNLOAD
70 MOVEXY SUM :CENTER: 1 FLOORPOSITION
80 LOAD
90 MAKE "ELEVATOR" SUM :GOTON: DIFFERENCE :ELEVATOR: :GOTOFF:
100 MAKE WORD "FLOOR" :FLOOR:
        DIFFERENCE SUM THING WORD "FLOOR" :FLOOR: :GOTOFF: :GOTON:
110 MAKE "X" BUTFIRST DIVISION RANDOM 2
120 TEST ZEROP :X:
130 IFTRUE MOVEDOWN
140 IFFALSE MOVEUP
150 SKIP
160 RUN
END
```

```
TO PEOPLE
10 TEST ZEROP :FLOOR:
20 IFTRUE STOP
30 MOVEXY :CENTER: FLOORPOSITION
40 TYPE "_"
50 MAKE WORD "FLOOR" :FLOOR: RANDOM
60 FOLKS THING WORD "FLOOR" :FLOOR:
70 MAKE "FLOOR" DIFFERENCE :FLOOR: 1
80 PEOPLE
END

TO FOLKS :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 TYPE "|"
40 FOLKS DIFFERENCE :N: 1
END
```

```
TO FLOORPOSITION
10 OUTPUT DIFFERENCE :BOTTOM: PRODUCT 2 :FLOOR:
END

TO SHOW :TIME: :POSITION:        ; displays the elevator
10 RESET :POSITION:
20 BOX
40 FOLKS :ELEVATOR:              ; and those in it
60 BOX
70 WAITM :TIME:                  ; for a little while
80 RESET :POSITION:
90 BLANKS SUM 2 :ELEVATOR:       ; then erases it
END

TO UNLOAD
10 MAKE "GOTOFF" RAND 0 :ELEVATOR:
20 GETOFF :GOTOFF:
END

TO LOAD
10 MAKE "GOTON" RAND 0 MINIMUM THING WORD "FLOOR" :FLOOR:
                SUM :GOTOFF: DIFFERENCE :CAPACITY: :ELEVATOR:
20 GETON :GOTON:
END

TO MOVEDOWN
10 TEST EQUALP :FLOOR: 1
20 IFTRUE MOVEUP
30 IFTRUE STOP
40 SHOW :QUICK: SUM FLOORPOSITION 1
60 MAKE "FLOOR" DIFFERENCE :FLOOR: 1
END

TO MOVEUP
10 TEST EQUALP :FLOOR: 10
20 IFTRUE MOVEDOWN
30 IFTRUE STOP
40 SHOW :QUICK: DIFFERENCE FLOORPOSITION 1
60 MAKE "FLOOR" SUM :FLOOR: 1
END

TO RESET :Y:
10 MOVEXY DIFFERENCE :CENTER: SUM 2 :ELEVATOR: :Y:
END

TO BLANKS :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 BLANK
40 BLANKS DIFFERENCE :N: 1
END

TO GETOFF :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 INSERTCHAR
40 TYPE  :
50 GETOFF DIFFERENCE :N: 1
END

TO GETON :N:
10 TEST ZEROP :N:
20 IFTRUE STOP
30 DELETECHAR
40 GETON DIFFERENCE :N: 1
END
```

```
                                  _:::                      _:::                      _:::

                                  _:::::                    _:::::                    _:::::

                                  _::::                     _::::                     _::::

                                  _:::::::::                _:::::::::                _:::::::::

                                  _:::::                    _:::::                    _:::::

                                  _::::::                   _::::::                   _::::::

                                  _:::::::                  _:::::::                  _:::::::

                                  _::                       _::                 ■:::■_:
                                                       ■::■
                         ■::::::■_                          _::::                     _::::

                                  _::::::                   _::::::                   _::::::
```

1.2 IMLAC$^{(R)}$

The IMLAC is a stand-alone computer with display capabilities which allow it to act as a local processor for Logo graphics commands. We first present two tables of the relevant Logo operations. Any abbreviations are included beneath the full operation names, the number of inputs (arguments) required is noted in parentheses with the description of each operation, and "$" indicates operations which were implemented after, and partly as a result of, the experiment discussed in this report. The first table describes the animation added after the experiment.

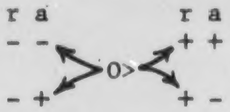
IMSSS Logo Animation Primitives$

ENDSNAP ENDS	(0-input operation) finish defining the current snapshot, and wipe the screen
ERASESNAP ERS	(1-input operation) escape from snapshot in progress (after SNAP, but before ENDSNAP) and reclaim drawing space and snap number; for snapshots which have been defined, ERASESNAP erases only the snapshot number (see WIPESNAPS)
MOVESNAP MOVS	(2-input operation) the first input is a sentence of "object" numbers (see PUTSNAP); the second input is a sentence of the relative distance and angle to move each object respectively (when "R" precedes the first object number, the sentence is interpreted as pairs of relative x,y distances to move each object); MOVESNAP returns a sentence of the new absolute x,y coordinates for each object; automatic wraparound occurs near screen boundaries (see WRAP)
PUTSNAP PUTS	(2-input operation) the first input is a sentence of pairs of snap and object numbers -- a zero object-number means create a new object; an object number between 1 and 100 may create or redefine an object; the second input to PUTSNAP is a sentence of pairs of absolute x,y coordinates for locating each object; PUTSNAP returns a sentence of the object numbers used (see MOVESNAP); to erase an object, redefine it to be an empty snap, i.e. snap 0 or any other undefined snap

SHOWSNAP SHOS	(1-input operation) show a snap (the input number) at the current turtle location
SNAP	(1-input operation) wipe the screen, and create a "snapshot" (the input number) out of the turtle commands that follow; presently ZAP does not work within snaps
WHATSNAPS WHAS	(0-input operation) return a sentence of the currently used snapshot numbers
WIPESNAPS WIPS	(0-input operation) wipe the screen and erase all snapshots and objects

IMSSS Logo Turtle Graphics and Plotter Primitives

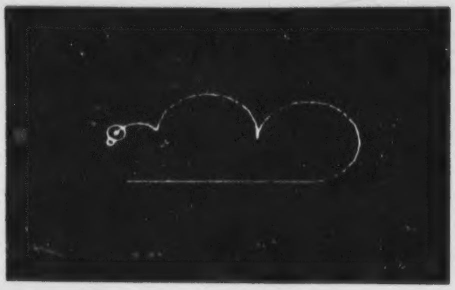
Some of these operations may be used to control both an IMLAC display and a Hewlett-Packard model 7202A XY plotter. The letter "P" indicates that an operation is implemented for the display and the plotter, while "*" indicates an operation that usually was not introduced to students.

ARC	P* (2-input operation) the first input is the radius (positive input makes the turtle go forward); the second input is the amount of arc to draw (positive is leftward); the chart shows the four different arcs for different signs of the inputs: 
ASETX	P* (1-input operation) move the turtle to absolute x (the input number), y remains the same
ASETXY	P* (2-input operation) move the turtle to absolute x (first input), y (second input) (see RSETXY)
ASETY	P* (1-input operation) move the turtle to absolute y (the input number), x remains the same
BACK	P (1-input operation) move the turtle back (the input number is the distance, see FRONT)
CLEAR	(0-input operation) clear the text area of the screen without erasing the turtle picture

COMPRESS	* (0-input operation) compress IMLAC display lists to recover more drawing space (only worthwhile if a picture has many short (less than 1 inch) lines); ZAP will not work on compressed pictures; if memory space is exhausted during a turtle command, the message: "DO YOU WANT TO COMPRESS THIS PICTURE? *" will appear; typing "YES" makes compression occur and drawing will continue if enough space is recovered
FRONT	P (1-input operation) move the turtle front along its current heading (the input number is the distance, same as BACK with a negative input)
HERE	P* (0-input operation) return the sentence of the turtle's current position and heading: x, y, angle
HIDE	(0-input operation) make the turtle disappear (see SEE)
HOME	P (0-input operation) move the turtle to home (see SETTURTLE); change only position and heading (not pen, head, or visibility)
LEFT	P (1-input operation) rotate the turtle left (counterclockwise the input is the number of degrees, same as RIGHT with a negative input)
PENDOWN PD	P (0-input operation) put the turtle's pen down so that when the turtle moves, it leaves a trace
PENP	P* (0-input operation) return "TRUE" if the turtle's pen is down, "FALSE" if it is up
PENUP	P (0-input operation) put the turtle's pen up so that when the turtle moves, it leaves no trace
PLOT	P (1-input operation) direct turtle commands to the plotter (input is the system's "tty" number for the plotter)
POKE	(0-input operation) poke out the turtle's head (see UNPOKE)
RIGHT	P (1-input operation) rotate the turtle right (clockwise, the input is the number of degrees, see LEFT)
RSETX	P* (1-input operation) move the turtle relative to its x position (the input is the x distance), y remains the same
RSETXY	P* (2-input operation) move the turtle relative to its x,y position (the first input is the x distance, the second input is the y distance) (see ASETXY)
RSETY	P* (1-input operation) move the turtle relative to its y position (the input is the y distance), x remains the same
SEE	(0-input operation) make the turtle appear (see HIDE)

SETHEADING SETHD	P* (1-input operation) set turtle to a specified angle (the input number is degrees)
SETSCALE	P* (1-input operation) set the display scale for turtle units units per inch (the input number)
SETTURTLE	P* (1-input operation) the input is a sentence of three numbers; the first number is turtle units per inch (see SETSCALE); the location of "home" is defined by the number of inches to the right (second number) and above (third number) the lower left corner of the screen, e.g. SETTURTLE "100 4 4" (the first turtle command causes this as default) sets a scale of 100 units per inch with home at the center of the screen
THERE	P* (1-input operation) the input is a sentence of three numbers: x, y, and angle (see HERE); equivalent to using ASETXY on the first two numbers, and SETHEADING on the third number, e.g. THERE "0 0 0" is equivalent to HOME
UNPLOT	P (0-input operation) direct turtle commands to the display, and release the plotter for others to use
UNPOKE	(0-input operation) pull in the turtle's head (see POKE)
WIPE	(0-input operation) clear the turtle area of the screen; move the turtle to home with PENDOWN
WRAP	P* (2-input operation) the first input is a sentence of the low and high x values to use for the screen boundaries (defaults are "-400 400"), the second input is similar for y; line clipping and wraparound (for MOVESNAP) occur at these boundaries
ZAP	(0-input operation) erase last turtle move with pendown or last series of consecutive moves with penup
ZIP	(1-input operation) the input is the number of turtle moves to ZAP

The IMLAC PDS-1 is a stand-alone computer with two processors. One processor is a general-purpose, 16-bit machine, the other is a display processor that refreshes the PDS-1's screen 40 times per second from a display list held in memory. Either processor may access memory by "stealing" instruction cycles from the other. The PDS-1 provides a screen resolution of approximately 96 points per inch. The screen can be viewed as a mesh of overlapping character "boxes", each approximately 5/8" square. Drawing lines across box boundaries requires either "long-vector hardware", or absolute or relative repositioning into an adjacent box before drawing can continue. Relative repositioning usually consumes more memory space, so we have used it only for drawing "snapshots", (i.e., translatable picture subroutines). The PDS-1s at IMSSS have only 4K of memory, and lack hardware for: multiplying, dividing, drawing long vectors, picture rotation, scaling, and translation. Furthermore, they use only one register for calls to display subroutines, which prevents the nesting of snapshots (newer models have "pushdown stacks"). To circumvent some of these difficulties, Sailogo compiles line segments, repositionings and picture subroutines into buffers of command words, word counts, addresses, and data (i.e., PDS-1-code display lists) and sends these over a 9600-bits-per-second line to a program (Graphics), running in the user's PDS-1, that realizes Logo's graphic abilities. The program also scrolls the user's typing and returns transmission-error ("checksum") information to Sailogo.* An organizational map of the PDS-1's memory, as defined by the Graphics program, appears on the following page; a sample Logo-graphics program follows that.



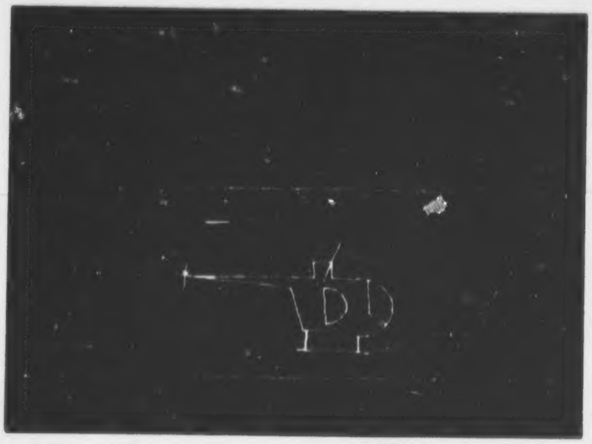
---

* We are grateful to John Prebus for his help with the IMLAC graphics programming.

```
executive program (Graphics) >------------------+
                                                |
buffers and text display lists                  |
                                                |
character display subroutines                   |
                                                |
user display space (2048 words)                 |
                                                |
                                                |
        turtle display subroutines              V
                                        -------------------------
  +---------->  head                     40 Hertz clock interrupt
  |                                      starts display program,
  | +-------->  pendown                  executive continues
  | |            V                      -------------------------
  | +-------->  penup                               V
  | |            V                                  |
  | | +------<  body                                |
  | | |                                             |
  | | | display position to home <--------------+
  | | |
+------<display jump to user picturelist
| | | |
| | | | display no-op (shows turtle) <--------------------------+
| | | |  |     or display halt (hides turtle) >----+            |
| | | |  V                                         |            |
| | +--<show pen (up or down) and body            --            |
| |   |                                         /    \          |
| |   +>display position to turtle head        | STOP |         |
| |                                             \    /          |
| +----<show head                                 --            |
|                                                 |             |
|       display halt (end of display program) >---+             |
|                                                               |
|       user display subroutines: <-------------------------+   |
|        "character mode" increments and relative           |   |
|        positionings and a display subroutine return >--+  |   |
|                                                        |  |   |
+-----> user picturelist:  a stack of "display cells"    |  |   |
         where a display cell consists of                |  |   |
                                                         |  |   |
                "character mode" increments              |  |   |
                and absolute positionings                |  |   |
                        or                               |  |   |
                an absolute positioning                  |  |   |
                        or                               |  |   |
                an absolute positioning and              |  |   |
                display subroutine jump >-------------------+   |
                                                         |      |
                followed by a display no-op <------------+      |
                        or                                      |
                display jump to show turtle (last cell) >-------+
```

The following is a listing of a Logo-graphics program which uses the IMLAC display to simulate a helicopter. The helicopter may be "flown" on the screen in response to a user's typing (e.g. "U" adds an upward increment to the helicopter's velocity, "H" makes it hover). The program was written by a student from the experiment discussed in this report. It is featured as part of the movie mentioned in Section 2.2.2. The student required some help to organize his ideas in terms of a sensible control structure. The procedure 'COPTER' initiates the simulation, 'FLY' maintains it and interacts with the user.

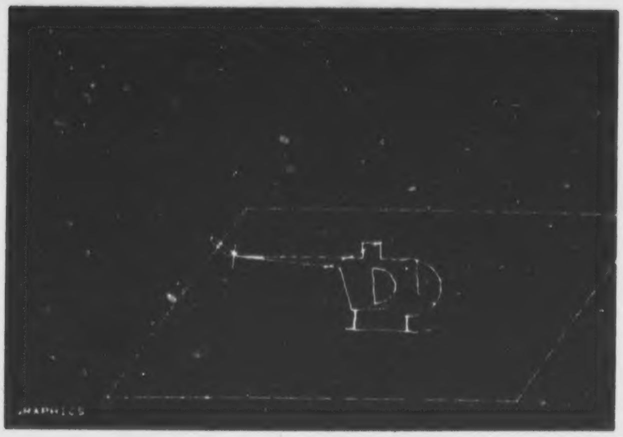
```
TO COPTER         ; initializes simulation and draws needed snapshots
5 WIPESNAPS
10 SNAP 1
20 FLYER
30 ENDSNAP
50 MAKE "XVEL" 0
60 MAKE "YVEL" 0
70 MAKE "RATE" 5
75 MAKE "GROUND" "TRUE"
80 SNAP 2
90 BLADE 240 0
100 ENDSNAP
110 SNAP 3
120 BLADE 150 30
130 ENDSNAP
140 SNAP 4
150 BLADE 60 70
160 ENDSNAP
170 SNAP 5
180 BLADE 30 90
190 ENDSNAP
195 SNAP 6
200 EARTH 370
210 ENDSNAP
220 SNAP 7
230 EARTH 300
240 ENDSNAP
250 SNAP 8
260 EARTH 200
270 ENDSNAP
272 SNAP 9
276 ENDSNAP
280 IGNORE PUTSNAP "1 1" "0 -250"        ; place the helicopter on screen

290 HIDE          ; turtle's work is done
295 CLEAR
300 FLY           ; take-off!
END
```



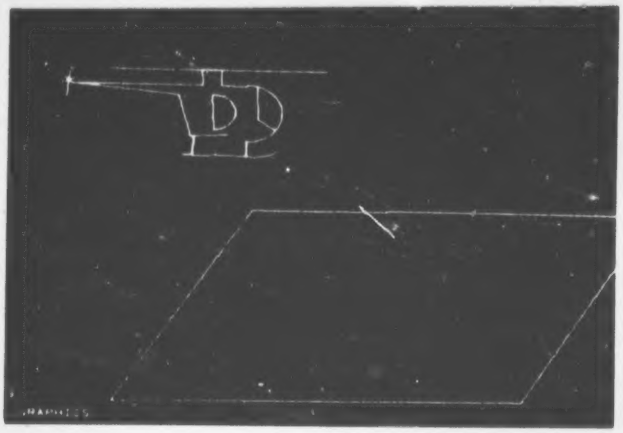
```
TO EARTH :SIZE:      ; draws views of ground
10 FRONT :SIZE:
20 RIGHT 120
30 FRONT :SIZE:
40 RIGHT 60
50 FRONT SUM :SIZE: QUOTIENT :SIZE: 2
60 RIGHT 120
70 FRONT :SIZE:
80 RIGHT 60
90 FRONT QUOTIENT :SIZE: 2
END
```

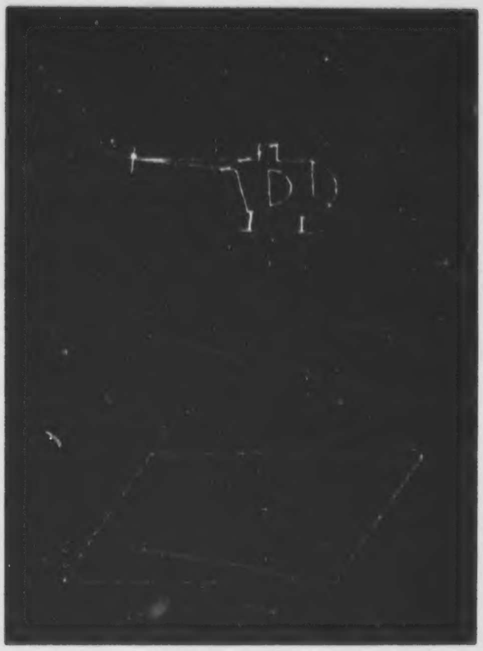
```
TO FLYER          ; draws the helicopter
1 PENUP
2 BACK 21
3 PENDOWN
4 FRONT 21
10 RIGHT 90
20 FRONT 20
30 LEFT 88
40 FRONT 30
50 PENUP
60 RIGHT 90
70 FRONT 80
80 LEFT 90
90 PENDOWN
100 ARC 40 180
110 PENUP
115 LEFT 90
120 FRONT 80
130 RIGHT 95
140 PENDOWN
150 FRONT 60
160 RIGHT 70
170 FRONT 60
180 LEFT 70
190 FRONT 110
200 PENUP
210 RIGHT 60
220 BACK 20
230 PENDOWN
240 FRONT 40
250 PENUP
260 BACK 20
270 LEFT 60
280 PENDOWN
290 FRONT 15
300 RIGHT 40
310 FRONT 30
320 RIGHT 60
330 FRONT 5
340 RIGHT 115
345 FRONT 40
350 LEFT 42
360 FRONT 140
370 LEFT 92
380 FRONT 20
390 PENUP
400 BACK 60
410 RIGHT 90
420 FRONT 10
430 LEFT 90
440 BACK 25
450 RIGHT 90
460 PENDOWN
470 ARC 25 180
480 LEFT 90
490 FRONT 50
500 LEFT 90
510 PENUP
520 FRONT 50
530 LEFT 90
540 FRONT 60
550 RIGHT 180
560 PENDOWN
570 FRONT 45
580 LEFT 50
590 FRONT 25
600 RIGHT 145
610 PENUP
620 FRONT 90
630 LEFT 90
640 FRONT 15
650 PENDOWN
660 FRONT 25
670 BACK 25
680 FRONT 25
690 LEFT 90
700 BACK 10
710 FRONT 70
720 LEFT 95
730 FRONT 20
740 BACK 20
750 RIGHT 95
760 FRONT 10
770 ARC 35 45
END
```

```
TO BLADE :LEN: :ROT:     ; draws rotor blades
10 PENUP
20 LEFT :ROT:
30 BACK QUOTIENT :LEN: 2
40 PENDOWN
50 FRONT :LEN:
END
```



```
TO FLY          ; manages the helicopter's motion
10 CHECK
20 MAKE "AIR" MOVESNAP "R1 2" SENTENCES :XVEL: :YVEL: :XVEL: :YVEL:
30 TWIRL 2 5
40 TEST :GROUND:
50 IFTRUE SNAPDOWN
60 IFFALSE SHOW.LAND
70 FLY
END
```





```
TO CHECK                       ; gives the "pilot" control
10 MAKE "KEY" ASKCHAR 0
20 TEST IS :KEY: "U"
30 IFTRUE MAKE "YVEL" SUM :YVEL: :RATE:
40 TEST IS :KEY: "D"
50 IFTRUE MAKE "YVEL" DIFFERENCE :YVEL: :RATE:
60 TEST IS :KEY: "L"
70 IFTRUE MAKE "XVEL" DIFFERENCE :XVEL: :RATE:
80 TEST IS :KEY: "R"
90 IFTRUE MAKE "XVEL" SUM :XVEL: :RATE:
100 TEST IS :KEY: "H"
110 IFTRUE MAKE "XVEL" 0
120 IFTRUE MAKE "YVEL" 0
130 TEST IS :KEY: "P"
150 IFTRUE MAKE "GROUND" "FALSE"
160 TEST IS :KEY: "G"
170 IFTRUE MAKE "GROUND" "TRUE"
END

TO TWIRL :BS: :ES:                   ; twirls the rotor
10 TEST GREATERP :BS: :ES:
20 IFTRUE STOP
30 IGNORE PUTSNAP SENTENCE :BS: 2 :AIR:
40 TWIRL SUM :BS: 1 :ES:
END

TO SNAPDOWN                                    ; makes the earth recede
10 TEST LESSP FIRST BUTFIRST :AIR: (-200)
20 IFTRUE IGNORE PUTSNAP "6 3" "0 -50"
25 IFTRUE STOP
30 TEST LESSP FIRST BUTFIRST :AIR: 200
40 IFTRUE IGNORE PUTSNAP "7 3" "0 -100"
50 IFTRUE STOP
60 TEST LESSP FIRST BUTFIRST :AIR: 400
70 IFTRUE IGNORE PUTSNAP "8 3" "0 -200"
END

TO SHOW.LAND                                  ; ground level
10 IGNORE PUTSNAP "9 3" "0 -350"
20 TEST LESSP FIRST BUTFIRST :AIR: AND - 150
40 IFTRUE MAKE "YVEL" 0
END
```

Appendix 2: Controllable Devices

In this appendix, we describe some electro-mechanical devices which IMSSS Logo now controls. Tables of the relevant Logo operations are included with the discussions of the devices. IMSSS Logo intelligently uses its knowledge of the type and location of any terminal that the user might pick. For instance, the physical Train may be controlled only from a terminal within sight of the layout; picking another terminal causes Logo to simulate the train instead.

2.1 Robot Turtle and Music Box

A controllable robot "turtle" is available from General Turtle Incorporated, Cambridge, Massachusetts. An interface allows the robot to be controlled by sequences of characters (ASCII) sent to it over a 30-characters-per-second line from Logo. The interface provides several "ports" to which one may connect turtles and/or "music boxes" for multiplexed (seemingly simultaneous) operation. The music box, as its name suggests, allows Logo programmers to write programs which play or generate musical compositions. It is an output device and returns no information to the controlling program. The turtle, on the other hand, does return some information (e.g. 'TOUCHLEFT') which allows one to write programs that adapt to the turtle's environment. In the following tables, any abbreviations are included beneath the full operation names and the number of inputs (arguments) required is noted in parentheses with the description of each operation.

IMSSS Logo Robot Turtle Primitives

Some operations have the same effect as for the IMLAC$^{(R)}$ graphics Turtle (see Appendix 1.2). "*" indicates that the operation usually was not introduced to students.

BACK	(1-input operation) move the turtle backward (the input value is the distance, see FRONT)
FRONT	(1-input operation) move the turtle forward along its current heading (the input value is the distance, same as BACK with a negative input)
LAMPOFF	(0-input operation) turn off the turtle's headlight
LAMPON	(0-input operation) turn on the turtle's headlight
LEFT	(1-input operation) rotate the turtle left (counterclockwise, the input value is the number of degrees, same as RIGHT with a negative input)
PENDOWN PD	(0-input operation) put the turtle's pen down so that when the turtle moves, it leaves a trace
PENP	* (0-input operation) return "TRUE" if the turtle's pen is is down, "FALSE" if it is up
PENUP	(0-input operation) put the turtle's "pen" up so that when the turtle moves, it leaves no trace
PLOT	(1-input operation) direct turtle commands to the robot turtle (input = -1, see UNPLOT)
RIGHT	(1-input operation) rotate the turtle right (clockwise, the input value is the number of degrees, see LEFT)
TOOT	(0-input operation) toot the turtle's horn
TOUCHBACK TB	(0-input operation) return "TRUE" if the turtle's rear sensor is touching something, otherwise return "FALSE"
TOUCHFRONT TF	(0-input operation) as for TOUCHBACK, but for front sensor
TOUCHLEFT TL	(0-input operation) as for TOUCHBACK, but for left sensor

TOUCHRIGHT (0-input operation) as for TOUCHBACK, but for right sensor
TR

UNPLOT (0-input operation) release the robot turtle for others to use

IMSSS Logo Music Box Primitives

The General Turtle music box is capable of producing sequences of synthesized tones from four simultaneous voices, as described below.

NOTE (2-input operation) the first input is a sentence of pitches; notes are buffered by the music system until the PM (play music) command is typed; pitches are specified by a decimal number between -28 and 32 or by a notation of the form (<octave>)<note> (<flat>|<sharp>), where:

<octave> is one of: (-2,-1,0 or <blank>,1,2)
<note> is one of: (C,D,E,F,G,A,B)
<flat> is < or -
<sharp> is #, > or +

Notation	Number	Tone
%, REST	-28	silence
BOOM	-27	"boom" percussion sound
SH	-26	"sh" percussion sound
	-25	(not used)
-2C	-24	C, two octaves below middle C
-2C#	-23	C sharp or D flat, same octave
-2D	-22	
-2D#	-21	
-2E	-20	
-2F	-19	
-2F#	-18	
:	:	
C	0	middle C
:	:	
2G#	32	G-sharp, two octaves above middle C

NOTE's second input is a sentence of pitch durations, which tell how long (e.g. how many 1/30th seconds) to sustain or send the corresponding note -- real-time duration thus depends on line speed and number of voices, so a duration of 30 for 1 voice lasts about 1 second -- equal to a duration of 7 for each of 4 voices (durations are between 1 and 127); for clear transitions between notes, a rest takes the place of the pitch at the end of the duration for durations >1; thus a duration of 1 sends the pitch once, 2 sends the pitch once and one rest, and so on

NVOICES	(1-input operation) the input is the number (between 1 and 4) of "voices" (independent, simultaneous note sequences) desired (3 voices is not recommended, see PM); it also erases any music already stored
PM	(0-input operation) play (and then erase) music that has been stored; voices which run out of notes (while others are still playing) are sent rests; (NVOICES 3 uses 4 voices, the fourth being entirely silent)
VLEN	(0-input operation) returns a sentence whose length is the number of voices; each word is the total duration for that voice
VOICE	(1-input operation) sends all subsequent notes to some voice (the input number is between 1 and 4, default is voice 1)

2.2 Train

One of the devices controlled by IMSSS Logo is an electric train. Here we present the relevant Logo operations followed by a description of the train system. Any abbreviations are included beneath the full operation names, the number of inputs (arguments) required is noted in parentheses with the description of each operation, and "*" denotes operations not normally introduced to students.

IMSSS Logo Train Primitives

BACK	(1-input operation) move the train backward a specific number of blocks (the input)
CONNECT	(1-input operation) connect a sentence of three locations, i.e. connect first and third locations by setting the switch at the second location
FRONT	(1-input operation) move the train forward a specific number of blocks (the input)
HOME	(0-input operation) move train to its starting location (see SETTRAIN)

SETSWITCH SW	(2-input operation) set switch (the first input) to some direction (the second input: "S" is straight, "C" is curved, "CL" is curved left, "CR" is curved right; * "DIS!" means disable switch in current position, "ENA!" means enable switch (first input can be "ALL"))
SETTRAIN	(1-input operation) set all switches straight; find the train (if at the terminal next to the layout), or begin a simulation by placing it at a starting location (the input word)
SPEED	(2-input operation) set the speed (second input (0,7), 0 means no change) for a direction ("F" or "B") and return the speed
TRAINFO	(1-input operation) return a sentence of information about a location (the input word), for example "" (i.e. not a location), "TRACK", "CROSSOVER", "SWITCH 2WAY STRAIGHT WORKING" and so on
TRMOVE	* (2-input operation) move the train to a block (first input) by the shortest route (approximately); if the second input is nonempty, TRMOVE just returns a sentence of route locations
TROP	* (3-input operation) general Sailogo operator; first input is the Sailogo command number, second and third inputs are parameters; returns :EMPTY: or result of operation; this is a way new train, turtle or graphics commands may be tested
WHERE	(0-input operation) return a sentence of locations; first (last) word is the next location if the train goes forward (backward) intermediate words describe where the train is; track location names are of the form ("*" indicates that the track is not clear): <track number> <zone letter> ("X" for crossovers); for example: 1F,2E,2EX are clear, *1F is a working disconnected switch, **1F is a nonworking disconnected switch, and *** is an end of track
WHERETO	(2-input operation) return a sentence of blocks accessible from given block (second input) in a direction away from the block named by the first input
WHISTLE	(0-input operation) train whistles (terminal bell rings)

The IMSSS train system differs from the basic Marklin$^{(R)}$ system in the following way: although the center rail is used for power, only one running rail is used for a ground return; the other rail is cut on block boundaries and is shorted to ground only when rolling stock with uninsulated axles enters a block. The interface signals that something entered or left block "xy", thus leaving to the program (Sailogo) the task of inferring what is that something. Supplying power to the engine via the center rail precludes independent control of a second engine unless a catenary or carrier-control system were added. A catenary would, of course, make the layout even more difficult to change. The interface's design allows for two trains (one via catenary), but this has not been exploited. Obtaining marginally reliable performance of one train is troublesome enough. For instance, speed control is rough and unpredictable. Sailogo can command eight different speeds, but only the highest causes movement, and even that slackens on curves. The interface also allows for coupling and uncoupling, but the physical problems involved have not been surmounted. Experiments in manually adding cars to the train indicate that excessive noise is induced in block sensing. Design of a reliable layout and control system requires imagination, experience in both electronics and model railroading, and plenty of time.* We feel that an initial simulation of train and layout would have led to better results.

Within Sailogo, knowledge about the track layout is incorporated in a linked list of track blocks (sections). Switches are blocks with added information about accessible, adjacent blocks. In setting a switch, the program checks that the switch is not occupied, replaces block links in the data structure, remembers the position of the switch (there is no hardware feedback on switch settings so all are initialized to be straight), and sends the appropriate ASCII character to throw the switch.

---

* We thank David Serres of Seattle for consulting with us on potentially reliable train designs.

When moving the train, the next block (forward or backward) cannot be a track end, be a disconnected switch, or be occupied. ASCII characters combining the train's direction and speed are sent to the train interface. To determine if the train is properly moving when feedback information is faulty (which can happen when track connections are dirty or loose), the program monitors a "window" of blocks, hopefully containing the train, its previous block and the next two blocks in its direction of motion. A transition matrix defines which of sixteen combinations of feedback information are stable, noisy, or erroneous (e.g. "0100" is a stable configuration where the previous block is unoccupied (0), the current block is occupied (1), and the next two blocks are unoccupied). For example, if the train enters a block not in the window, a faulty switch may be the cause (the program could, but does not, incorporate this knowledge into its world view); if the train skips a block (e.g. 0001), a track sensor may have failed and this fact is reported; when the train does not change state within a few seconds, a blown circuit-breaker or broken connection may be the culprit; and so on. In this way, Sailogo attempts to ignore noise and classify and compensate for errors where possible.

If we could have added multiple trains (either physically or in a simulation), there remain problems with the Tenex control structure in allowing separate users to share a device and/or a changing, program data-base. As one solution, we added multiple Logo users as subsidiary

Tenex forks to one train job, which then acted as an executive to allow train commands to be executed in a shared environment. This approach was frustrated because Logo lost the ability to handle pseudo-interrupts unless they were generated by the additional terminals (in the current version of Tenex (1.31) at IMSSS, there may be but one controlling terminal per job).

Appendix 3: Details Pertinent to the Preliminary Test

This appendix contains a discussion of how one commercial designer of an aptitude test for computer programming ability attempted to assess the validity of that test. The test was examined in the process of designing our own test -- a sampling of which appears in Appendix 3.2.

3.1 An Example of Commercial Evaluation

The example derives from remarks in the published manual for one of the tests we investigated. The validity of that test was assessed by three studies: (1) correlation of test scores and grades of three groups of programming trainees, (2) correlation of test scores and overall performance ratings by supervisors of programmers, and (3) a study like that of (2) in which grades on a training course were also available. Studies (1) and (3) both assumed, without discussion, that the testing done during training was itself a valid measure of programming ability. Studies (2) and (3) both assumed that ratings by superiors was similarly valid. Study (1) indicated that, of fifteen relevant correlations between subtest scores and trainee groups, eight were of statistical significance. And only one subtest was significantly correlated with trainee performance over all groups, in spite of the fact that the overall test/training correlation for each group was significant. Interestingly, the most variable subtests were those which relied heavily on time and repetition. In Study (2), three of five subtest correlations and the overall correlation were significant but small; and the two remaining subtests were those which exhibited variable or minimal correlation with performance in study (1). Unfortunately, the ratings used as the validating measure in (2) were not confined to programming ability and included such things as attitudes. Therefore, study (2) is invalid. Study (3) found three subtests significantly correlated with training course grades, but one of the three had not been significantly correlated with grades for any group in study (1). Furthermore, the ratings used in the other half of study (3) were virtually uncorrelated with subtest results. The brochure went on to state that these ratings and job tenure were correlated more strongly than anything else in both halves of the study -- the suggestion being that low correlations must be expected when evaluations place high value on relatively invalid properties (i.e. tenure). An alternative observation can be made which applies to any correlational procedure: the sample variance of a measured property may be so low that apparent but spurious correlations with another measure arise. In study (3), the test scores could have had low variability for good reason: the testees could have been of very nearly the same competence. In any event, none of the studies provided a clear validation of this particular test for programming aptitude.

3.2 Sample Problems from the Preliminary Test

.....

here are some boxes with numbers in them. Each box also has a permanent name of its own, which is also a number.

1	2	3	4	5	6	7	8	9	10
6	3	9	2	11	2	91	48	66	1

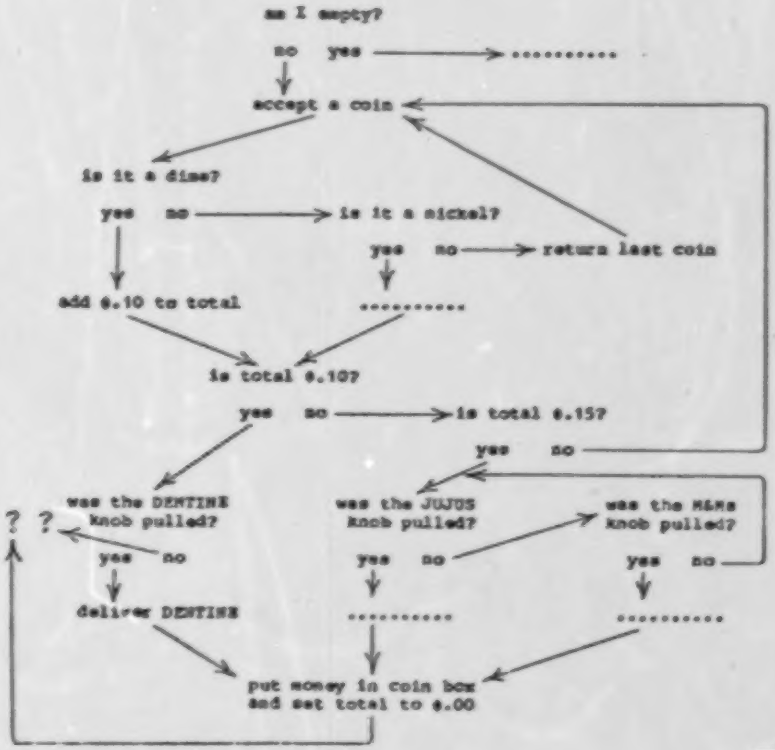
imagine that you are a robot that can read and write. Please obey these commands (note, when you write in a box, you replace what it held before):

1. add the number in box 4 to the number in box 2
   and write the sum in box 7.
2. add the number in box 7 to
   the number found in the box whose box number is in box 6
   and write the sum in box 6.
3. multiply the number in box 6 by the number in box 1
   and write the product in box 5.
4. stop being a robot.

what number is now in box 5?

.....

here's a diagram that shows how our candy machine works, please start at the top, follow the arrows and fill in each blank with what you think the machine should do at that point:



.....

try to spell as many 4-letter words as you can from the letters:

BANG

.....

if you ask for change of three $1 bills in dimes and nickels and you get twice as many nickels as dimes, how many nickels will you have?

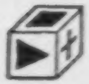
.....

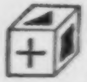
using you knowledge of the alphabet, what do you think the next two letters in this sequence should be?

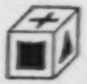
ghgfghg

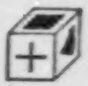
.....

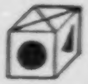
imagine that you can rotate the cube on the left as much as you like. Which one cube on the right is probably the same as the cube on the left?











.....

please fill in the blanks to make a reasonable sentence:

...... is to evening as breakfast is to ....

.....

what is the one simple rule which allows each lefthand word to be changed into each righthand word?

CALENDAR	LACENDAR
SUMMER	MUSMER
TREE	ERTE

.....

here are the definitions of some ordinary mathematical symbols which you may have seen before:

symbol	meaning
>	is greater than
<	is less than
=	is equal to
≠	is not equal to
≯	is not greater than
≮	is not less than
IF . THEN ..	if we know . is true then .. is also true

so 3 < 4 is a true statement about the numbers 3 and 4, but X < Y may be true or false depending upon what numbers are put in place of X and Y, but IF X > 1 THEN 0 < X is a true statement for any number X.

in each of the mathematical statements below, A, B and C stand for "any number", for each statement, please say whether you think it is:

ALWAYS TRUE,
ALWAYS FALSE,
or SOMETIMES TRUE OR FALSE.

statements:

IF A > B > C THEN A = C

IF A = B = C THEN A ≯ C

IF A = B ≠ C THEN A ≠ C

IF A ≠ B ≠ C THEN A ≠ C

IF A ≮ B ≠ C THEN A ≮ C

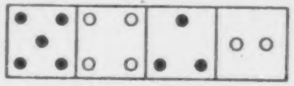
.....

Baskina the robber broke into an ice cream factory and lifted a four-foot cube of vanilla ice cream, but was surprised by the police in her escape and fell with the cube into a vat of chocolate syrup. The police took the ice cream for evidence but couldn't carry it down to the station in one piece, so they cut it into one-foot cubes, packed in dry ice to prevent melting.

how many one-foot cubes of ice cream had no chocolate on them?

.....

here is a sequence of four figures, they change from left to right according to a simple rule, try to discover it and draw the next figure in the sequence:



please write, in a few words, what you think the rule is:

.....

here is an addition problem in arithmetic:

$$\begin{array}{r} A\ A \\ +\ B\ B \\ \hline C\ B\ C \end{array}$$

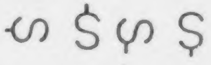
A, B and C are three different digits between 0 and 9.

A = ... B = ... C = ...

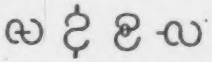
.....

the figures in group 1 have something in common that is not shared by any of the figures in group 2. circle the figure in group 3 that belongs in group 1 but not in group 2.

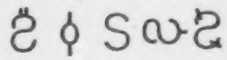
group 1



group 2



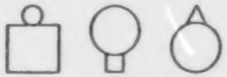
group 3



describe the difference between figures in group 1 and group 2.

.....

figure A was changed into figure B by a simple rule, please draw figure D so that it corresponds to figure C changed by the same rule:



A B C D

what is the rule in words?

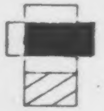
.....

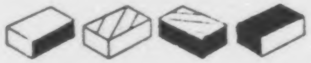
what one simple rule, not using arithmetic, was used to make the digits on the right from the strings of digits on the left?

999999999	9
556	5
6106	6

.....

which of the boxes on the right could have been made with patterns like that on the left:





.....

if . => .. means "replace . with .. ", we can change the word FOG into the word DOG by using these rules on FOG :

F => D OG.=> OG

please try to make up some rules of your own which can change SUMMER into WINTER :

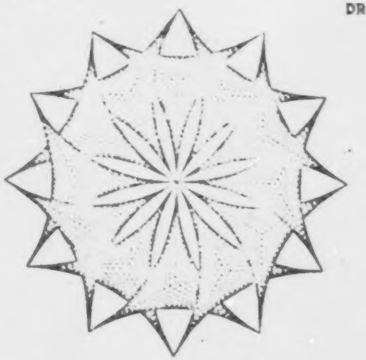
Appendix 4: Sample Logo Curriculum

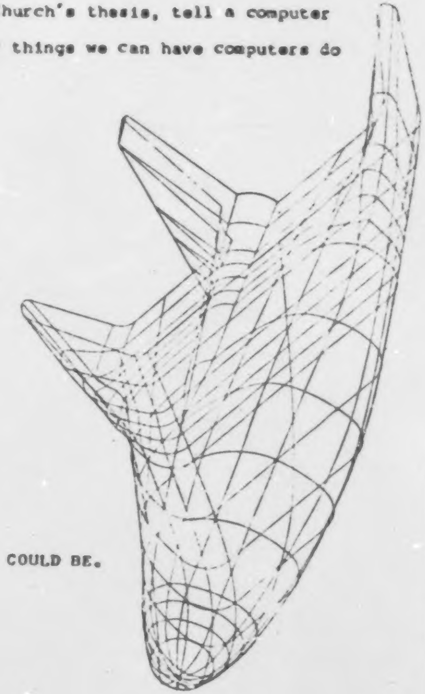
This appendix contains portions of the Logo curriculum developed for the experiment discussed in this report. Many of the excerpts shown here are referenced by discussions in the text, particularly in Sections 4.1 and 6.1. The following table indexes the curriculum by part number, curriculum page, and page of this appendix. Pages denoted with "T" were designed for use by turtle-graphics students. The text is copyrighted, but may be used for noncommercial purposes.

Part	Curriculum Page	Page
1	1,4	173
2	8	174
3	15,17T,17.2T	174-175
4	18,21,22	176-177
5	26-30	177-179
6	34,40-42,44T	180-182
7	46,47,50,54,54T	182-184
8	55,59,61,63	185-186
9	65,66,68,70,72,74,75	187-190
10	76-78,79T,80T,81,82,84,85	190-194

All the things above can be done by we humans and we can show another human how to do them, so we can, by Church's thesis, tell a computer how to do them. Here are some other things we can have computers do for us:

DRAW





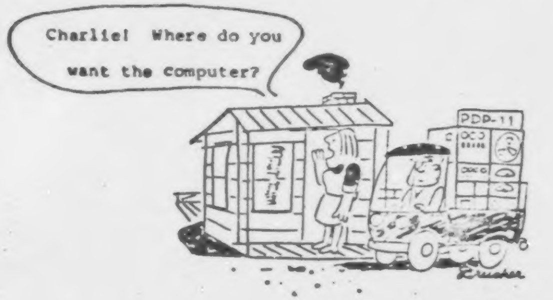
PLAY GAMES

I AM THINKING OF SOMETHING THAT YOU COULD BE.
THE WORD HAS 6 LETTERS.

GUESS A LETTER:

Part 1

Computers are machines invented by people to help people. A computer is a tool which anyone can learn to use in any way he or she likes.


Charlie! Where do you want the computer?
PDP-11

An ordinary computer is so useful that, as far as anyone knows, it can be used to solve any problem or puzzle that any human can solve. This strong statement means that computers are the most general tool people have yet invented.

Do you know what a TOOL is? (please follow the arrow from your answer)

YES ↓

NO ↓

A hammer is a tool. So are racing cars, books, trumpets, and even toothbrushes. A TOOL is "anything used to perform some operation or achieve some result, for work or pleasure".

Please write down the names of a few tools:

Copyright 1974, A. B. Cannara and S. A. Weyer.

JOHN65b and 123ABC also contain characters like , and letters of the alphabet which aren't numbers and must be quoted if we want them to be literals.

"MARY is not a literal (or anything else in the Logo language because it is missing the right quote mark.

You can tell Logo to PRINT a value on your teletype or Imlac.

```
PRINT "MY NAME IS"
P "A SHORT SENTENCE"          (Logo understands P to mean PRINT,
                              this is called an "abbreviation".)
P 123456789
P "LOGOWORDSDONOTHAVESPACES"
```

Try the following to find out what Logo does (remember to end each command with the RETURN key).

```
PRINT 100                    P "XYZ"
P XYZ                        P "XYZ" "XYZ"
P "TIME"                     P TIME
PRINT                        P "100"
```

Make up some literals and command Logo to PRINT them.

Type:

```
PRINT "A VERY VERY VERY LONG SENTENCE"

then by typing      control control control control control control
the control keys:      N       N       N       N       B       E

Logo will
type for you:      PRINT    "A     VERY    VERY           LONG SENTENCE"
```

Copyright 1974, A. B. Cannara and S. A. Weyer.

Now that you have Logo's attention, type some sentence, anything you please, with more than one word in it. What did you type? ....

Now tell Logo that you are finished with the sentence by typing the RETURN key (CR on an Imlac). Logo will probably reply "...... NEEDS A MEANING". What did it reply? .... What word in the sentence you typed did Logo mention in its reply to you? .... Try some other sentences (we will call them "lines").

HI, THERE! MY NAME'S MARK I'M THE NEW OPERATOR!
HI MARK MY NAME'S TED!
MARK I THINK YOU'LL LIKE THE PDP-11! IT'S A GOOD LITTLE COMPUTER A NICE MACHINE WITH AN EVEN TEMPERAMENT!
TREAT THE PDP WITH RESPECT, AND I THINK YOU'LL FIND IT DELIVERS! O.K., I'VE TURNED IT ON! IT'S YOUR BABY NOW MARK. GO TO IT! TAKE HER OUT FOR A SPIN!
TAKE HER OUT FOR A SPIN?
GO WILD SON!
7-4

Although Logo did not understand the English words you typed, do you think it's strange that Logo answered in English? .... Logo only understands certain English words and expects them to be arranged in certain orders. For example, type BELL followed by the RETURN key. In a little while you will find out which words Logo understands already and how you can make Logo understand other words.

Whenever you type a semicolon ";", Logo ignores everything you type after that on the line.

```
* ;I DARE YOU TO ANSWER ME, LOGO!!!!
*
```

182

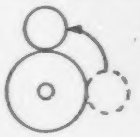
What do you think the WIPE command does to a picture? .... Try it.

Wi

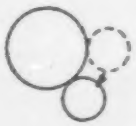
Control-X does to a sentence what .... does to a picture.
.... does to a word what ZAP does to the last line drawn.

If the turtle always moves with its PENUP, will you ever see a picture?

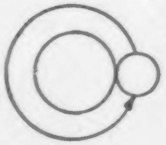
You may be getting tired of moving and drawing with the turtle in the same direction. You can turn the turtle. Type LEFT or RIGHT followed by one input which is the number of degrees to turn. It takes 360 degrees for the turtle to completely turn around so that it is headed in the same direction. Ask a tutor if you don't know what "degrees" means (the turtle does not get warmer or colder).



LEFT 90 or
RIGHT -90



RIGHT 45 or
LEFT -45



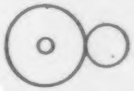
LEFT 360 or
RIGHT 360

Try some of the following examples or make up your own:

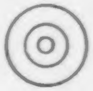
[✓ LEFT SUM 45 AND 90] ✗ RIGHT "DEGREES"
✗ LEFT ✗ RIGHT REQUEST
[✓ LEFT QUOTIENT OF 360 AND 6] ✗ RIGHT LEFT 10

After moving and turning the turtle, type HOME. This will return the turtle to the original direction and position it had when it first appeared on the screen. (WIPE does this too, but it erases your picture and puts the PENUP).

If your Imlac terminal does not say GRAPHICS in the lower left corner, ask a tutor for help. Besides PRINTing things on part of the Imlac screen (this is called the "typewriter" part), you can command the Logo "turtle" to move around the screen and draw pictures. The turtle can POKE its head out and also UNPOKE it (pull it in). Is this like a real turtle? .... This is what the turtle might look like at various times.



POKE



UNPOKE

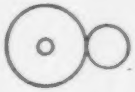
HIDE

Any time you type SEE, the turtle will appear. If you type SEE when you first start Logo, the turtle will appear at the center of the screen. Please type SEE (you may have to wait a few seconds for the turtle to wake up.)

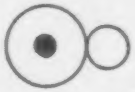
Is the turtle's head POKEd or UNPOKEd? Poked ....

Try each of the commands UNPOKE, POKE, HIDE and SEE until you are sure of what each command does to the turtle's appearance.

The turtle has a pen to draw with, which can be up or down.



PENUP
(and POKE)



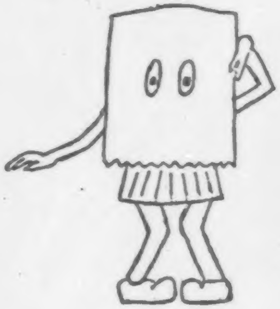
PENDOWN
(and POKE)

The turtle on the screen has its pen in a PenUP .... position.

Copyright 1974, A. B. Cannara and S. A. Weyer.

The value of a name is not obvious from looking at the name. You have to use the THING command to find out the name's value. Luigi Literal (see page 13LT) has nothing up his sleeves and you can see his value immediately.

She's hiding something.



Naomi Name

Naomi Name is hiding her value but she will tell you her value if you use the THING command. You can use the MAKE command to make her remember a value for you.

Type the following MAKE commands:

```
MAKE "SNOOPY" "CHARLIE BROWN"

MAKE "CHARLIE BROWN" "LINUS"
(since a name doesn't have to be a literal,
you could type MAKE :SNOOPY: "LINUS")

MAKE "LINUS" "BLANKET"

MAKE "DISTANCE" 25

MAKE "ANGLE" PRODUCT 3 AND THING OF "DISTANCE"
(this can be shortened to MAKE "ANGLE" PROD 3 :DISTANCE:)

MAKE "25" "TWENTY FIVE"

MAKE "2" "3"

MAKE "ANSWER" REQUEST
```

Part 4

In part 3, you probably noticed the sneaky introduction of two new commands, SUM and WORD. SUM takes two inputs (PRINT only had one!) adds them together (so they must be numbers) and outputs their sum (the command SUM has the value of the sum of the two inputs). WORD takes two inputs. It glues the beginning of the second input to the end of the first input and outputs a new word. Try:

```
    2 P SUM OF -3 AND 5              ABC77 P WORD "ABC" 77
    X P "ABC" 77                    111222 P WORD "111" "222"
    X P SUM 2                            X P WORD "AWORD" "A SENTENCE"
   12 P SUM SUM 3 4 5               LOGCGO P WORD WORD "LO" AND "GC" "GO"
XYZ20 P WORD "XYZ" AND SUM 9 AND 11    102 P SUM 3 AND WORD OF 9 AND 9
```

You can decide for yourself whether or not you want to use OF (allowed between the name of a command and its first input) and AND (allowed between a command's inputs). Do you think using AND and OF makes Logo look more like English? Yes ....

So far you have used values in the form of literals (which output themselves) and commands like WORD and SUM which output values depending on their inputs. The command REQUEST does not have any inputs but makes Logo wait for you to type something on the typewriter and outputs that value. You can command Logo to MAKE a "name" have some value (Logo calls this value a "thing") which is a literal or output of some command. For example,

```
MAKE                           (what you would type is underlined.)
----
  NAME:    "DIGITS"            (a shorter way to say the same command
           --------
  THING:   "0123456789"           is MAKE "DIGITS" "0123456789" )
           ------------              -----------------------
```

Copyright 1974, A. B. Cannara and S. A. Weyer.

Part 5

Type: RECTANGLE

Does Logo understand this command? NO ....

What do YOU do when you don't understand a new word? LOOK it UP ....
IN THE [illegible] ....

You can tell Logo what a new word means by defining it with words Logo already knows. Is this like looking up a word you don't know in the dictionary? Yes ....

To define your own Logo commands (these are called "procedures") type TO, a space, and the name you want your procedure to have. Logo now responds with a # (instead of a ← to show that you are defining a procedure. You can type commands just as before and Logo will obey them. If you want Logo to remember a command as part of your procedure, then you must precede the line by a "line number" and a space.

Please type the underlined words.

←TO RECTANGLE

#PRINT "***" (Logo obeys this immediately but only remembers it while it is the last line you typed.)

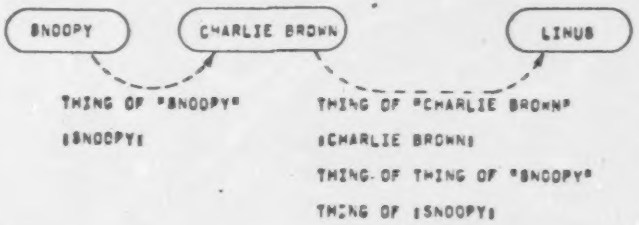
#10 P "+++" (Logo does not obey these commands now but remembers them for later.)

#20 P "+++"

(What could you type now to make Logo obey the command on line 20? Ctrl(S)&(E) .... (Hint: two control- commands) Control-S. erases the line number from the line Logo is remembering (line 20). control-E, types the rest of the line. If you don't understand this, ask a tutor for help.)

My name has a value!!?

What is the value of (hint: try PRINTing the values)	???????
"SNOOPY"	SNOOPY
:SNOOPY:	CHARLIE BROWN "
THING OF :SNOOPY:	LINUS
THING OF THING OF "SNOOPY"	LINUS
THING OF "BLANKET"	....
THING OF 25	TWENNY FIVE
:DISTANCE:	25
THING OF "CHARLIE BROWN"	LINUS
THING OF :CHARLIE BROWN:	BLANKET
"2"	2
THING OF 2	3
THING OF SUM OF 19 AND 6	TWENNY FINE
THING OF WORD OF "LI" AND "NUS"	BLANKET
THING OF "ANSWER"	"SNOOPY



Copyright 1974, A. B. Cannara and S. A. Weyer.

```
*EDIT RECTANGLE
 --------------

#25 P "END OF RECTANGLE"              (You can add new lines.)
 ----------------------

#25 P "THERE WENT A RECTANGLE"        (You can replace lines.)
 ----------------------------

#EDL 5          (This is short for EDIT LINE 5.  Logo now makes line 5
 -----
                the previous line typed, so you can use control-E,N,S.)

                control  control control               control     RETURN
                   N        N       S                     E

5               PRINT     "HERE              COMES    A RECTANGLE"
                                             ------

#END                            (You've finished redefining RECTANGLE)
 ---

*LIST RECTANGLE           (To see that Logo remembers your procedure,
 --------------
                           type LIST followed by the procedure name.)
TO RECTANGLE
5  PRINT "HERE COMES A RECTANGLE"
10 PRINT "***"
15 PRINT "* *"
20 PRINT "***"
30 PRINT "THERE WENT A RECTANGLE"
END

*RECTANGLE
 ---------

HERE COMES A RECTANGLE
***
* *
***
THERE WENT A RECTANGLE
```

When Logo starts to follow the instructions you have typed in RECTANGLE, it begins with the lowest line number, which is line ----- in RECTANGLE. Logo continues obeying commands in order of increasing

```
#5 P "HERE IS A RECTANGLE"      (You can insert lines out of order and
 -------------------------
                                 between other lines-- it's a good idea
#15 P "* *"
 ----------                      to leave some room by numbering lines

                                 by 10's)

#LIST                            (This shows you what Logo remembers
 ----
                                 so far for RECTANGLE.  Logo types your

TO RECTANGLE                     procedure in the order of its line

5  PRINT "HERE IS A RECTANGLE"   numbers and expands abbreviations.)

10 PRINT "***"

15 PRINT "* *"

20 PRINT "***"

#END                             (You've finished defining RECTANGLE.)
 ---
```

To make Logo obey RECTANGLE, just type its name.

```
*RECTANGLE
 ---------

HERE IS A RECTANGLE
***
* *
***
```

Since RECTANGLE needs no inputs (values) and has no output (value), it may be represented as just:

Besides editing lines (which you should know how to do now -- remember control-A,E,N,R,S,W,X??), you can also edit procedures. Type EDIT followed by the procedure name and the RETURN key.

Copyright 1974, A. B. Cannara and S. A. Weyer.

Type RECTANGLE. To get Logo to stop RECTANGLE, type control-G. Your RECTANGLE procedure is like the story: Pete and Repeat were in a boat, Pete fell in, who was left? If you say "Repeat", then you have to listen to the story again and again and again. Do you think RECTANGLE will ever stop? ....

Here is a procedure for the Pete and Repeat story.

```
        TO REPEAT
        10 PRINT "PETE AND REPEAT WERE IN A BOAT."
        20 PRINT "PETE FELL IN.  WHO WAS LEFT?"
        30 TEST IS REQUEST "REPEAT"
        40 IFFALSE PRINT "I'M SORRY, IT WAS REPEAT"
        50 REPEAT
        END
```

Write a procedure to draw something with the turtle. Try to do it alone first. Talk to a tutor if you can't think how to begin. Some ideas are to draw a rectangle, a triangle, some letters of the alphabet or to make the procedure below do something else. Your procedure should not be more than ten (10) lines long. You can divide the problem into several procedures and then use their names in a "start" procedure.

```
        TO MEET          ; LOGO IGNORES STUFF AFTER SEMICOLONS
        10 PRINT "I'M THE TURTLE.  WHAT'S YOUR NAME?"
        20 MAKE "YOURNAME" REQUEST
        30 PRINT SENTENCE "HOW FAR DO YOU WANT ME TO MOVE, " :YOURNAME:
        40 PENDOWN
        50 FRONT REQUEST
        60 PRINT "THINKING AWHILE..."
        70 WAIT 10                    ; LOGO WAITS 10 SECONDS
        80 PRINT "SCREECHING AROUND A CORNER"
        90 LEFT 90
        100 FRONT 100
        END
```

You can ERASE a procedure with the ERASE command. For example, ERASE RECTANGLE. What will happen now if you type LIST RECTANGLE? ....

Control-R does to a sentence what .... does to a procedure.

.... does to a sentence what ERASE   does to a procedure.

WIPE   does to a picture what .... does to a procedure.

line numbers. After Logo obeys the command on the highest numbered line (which is ......), it stops and goes back to obeying (giving control to) whoever last used the name RECTANGLE. Since you typed RECTANGLE as a command from the typewriter, Logo returns to the typewriter for more commands to obey. Do you think that you can use a procedure as a command on a line in another procedure? ....

```
Try:    TO TWORECTANGLES

        10 RECTANGLE    (We say that TWORECTANGLES gives control to

                        ("calls") the RECTANGLE procedure.  Logo obeys

                        RECTANGLE until RECTANGLE stops.  Then Logo goes

           ------------ back to obeying TWORECTANGLES here.)
           |
           V
        20 PRINT "OH, NO, NOT ANOTHER ONE!"

        30 RECTANGLE              (TWORECTANGLES calls RECTANGLE again.)

        END
```

Change TWORECTANGLES so that line 15 is STOP. (Look at page 28LT if you've forgotten how to EDIT a procedure.) Now type TWORECTANGLES. Did Logo print the second rectangle? .... So you can use the .... command to make a procedure stop before its last line.

Do you think that you can use a procedure as a command on one of its own lines (call a procedure from itself)? .... Change RECTANGLE so that line 40 is RECTANGLE.

```
        TO RECTANGLE
        5  PRINT "HERE COMES A RECTANGLE"
        10 PRINT "***"
        15 PRINT "* *"
        20 PRINT "***"
        30 PRINT "THERE WENT A RECTANGLE"
        40 RECTANGLE
        END
```

Copyright 1974, A. B. Cannara and S. A. Weyer.

Change DOUBLE as follows (type line 10 exactly; as it is):

```
EDIT DOUBLE
10 OUTPUT SUM :NUMBER: :NMBER:  ; a bug because of misspelling
END
```

Type: P DOUBLE 7

Did Logo complain? ....   Is it a bug? .... Why? ....

There is usually more than one way to write a procedure that does the same thing. Change the DOUBLE procedure so that it doesn't use the SUM command. (Be sure to fix the misspelled :NMBER:). Show a tutor that your new DOUBLE procedure works.

Write a procedure to UNDOUBLE a number (it should OUTPUT rather than PRINT). Ask a tutor if you need help.

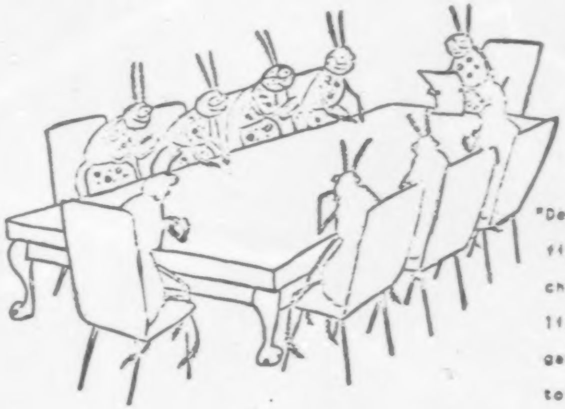
When you type	what do you get?
P UNDOUBLE DOUBLE 3	3
P UNDOUBLE DOUBLE 4	4
P UNDOUBLE DOUBLE 999999	999999

Does UNDOUBLE undo what DOUBLE did to a number? yes

When you type	what do you get?
P DOUBLE UNDOUBLE 8	8
P DOUBLE UNDOUBLE 7	6
P DOUBLE UNDOUBLE 999	997

Does DOUBLE undo what UNDOUBLE did to a number? almost

Sometimes a procedure of yours may not work the way you thought it would (we say there is a "bug" in your procedure). Logo may give you a message (like THERE IS 1 INPUT MISSING FOR SUM or QUZ NEEDS A MEANING) or Logo may not do the things you expected (like making rectangles instead of triangles, or never stopping -- control-G stops a "runaway" procedure). These "bugs" are not the kind of insects studied by entomologists nor are they electronic listening devices. Although bugs may look like mistakes when you first see them, you will learn to see their "ant"-ics as funny.



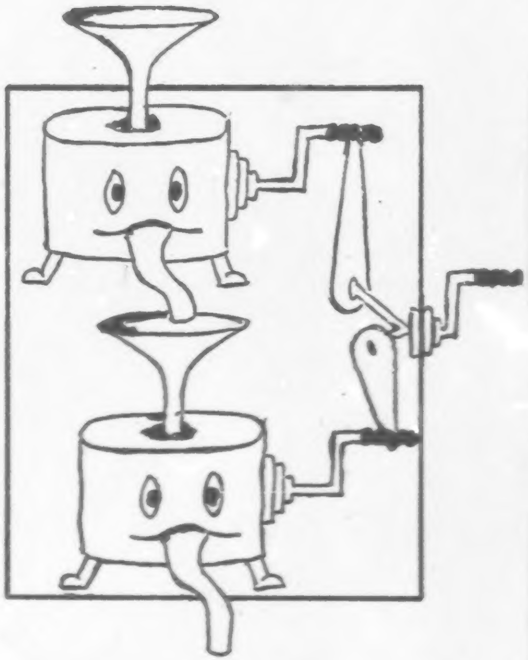
Bugs! What can we do to their procedures?

"Debugging" (the process of finding, understanding, and changing bugs) is very much like playing a detective game like Clue. You need to know who did what, where, and why.

For example, SUM OF "" AND "3" should make Logo complain: INPUTS MUST BE NUMBERS. It might be a clue to what is wrong. A similar message: I WAS AT LINE 30 IN POIUYT might tell you where to start hunting for bugs in POIUYT. If you have no idea where to start, then begin at the beginning, with the first procedure you asked Logo to obey.

Copyright 1974, A. B. Cannara and S. A. Weyer.

when you connect the spout of one function to the funnel of another function, (for example, DOUBLE SUM 3 4 or UNDOUBLE DOUBLE 77) this is called function "composition". The new function is really composed of two or more parts which are functions themselves.



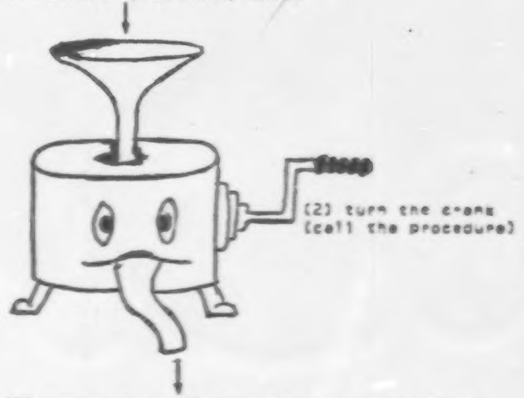
Function Composition

A function is a "partial" function if it does not work for all inputs. For example, DOUBLE "ABC" will give the DOUBLE function machine indigestion because it does not know what to output (Logo will give you the message ....)

Why doesn't DOUBLE UNDOUBLE work for 7 or 999? .... For what other numbers will it fail? ....   Is there a rule? ....

DOUBLE and UNDOUBLE are "functions" because they take one or more inputs and give you an output. You can (1) drop some inputs into the funnel of the function machine (procedure) below

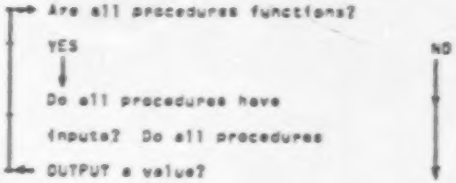
Funky Function



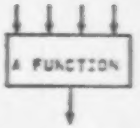
(2) turn the crank
(call the procedure)

and (3) get your output from the spout (spoutput).

Functions always have one or more inputs and one output.



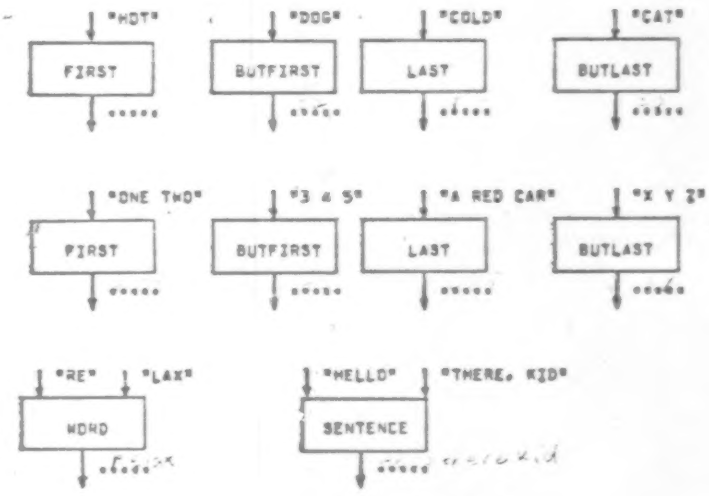
Are all procedures functions?
YES
Do all procedures have inputs? Do all procedures OUTPUT a value?
NO



Copyright 1974, A. B. Cannara and S. A. Weyer.

when you type	what do you get?
P FIRST "BUG"	B ....
P BUTFIRST "A VERY VERY LONG SENTENCE"	[illegible] ....
P LAST "1 2 3 WHAT ARE WE FIGHTING FOR"	for ....
P BUTLAST "SLOPE"	slop ....
P BF BF BF BF BF "ELEPHANT"	ant ....
P BL BL "THIS IS TOO MUCH TO TYPE"	[illegible] ....
P F BF BF "COOKIE MONSTERS LOVE COOKIES"	love ....
P BL L "THE FATE OF THE FREE GALAXY"	galaxy ....
P BF F "UNIDENTIFIED FLYING OBJECTS"	nidentified ....
P SENTENCE OF "THIS IS" AND "FUN?"	[illegible] ....
P S S "I LIKE HAMBURGERS" "WITH" W "KET" "CHUP"	[illegible] ....

Fill in the missing results.



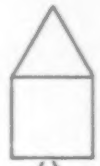
Write a procedure with two inputs which makes the turtle draw a rectangle. One input should be the length of the rectangle, the other should be its width. For example,

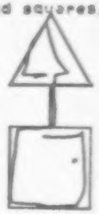
```
TO RECT :LENGTH: :WIDTH:
; you fill in the rest
END
```

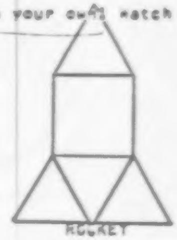
Write a procedure with one input which draws a square. The input should be the length of a .... (You can write the SQUARE procedure so that it just calls on the RECT procedure and RECT does all the work for you.)

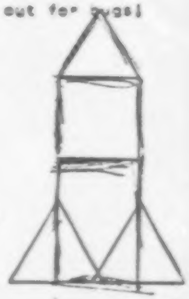
Write a procedure with one input to draw an equilateral (all sides equal and all angles equal) triangle. The input should be the length of a side.

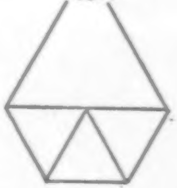
Write procedures to draw some of the pictures below made out of triangles, rectangles, and squares. Design your own; watch out for bugs!

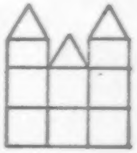

HOUSE


WISHING WELL


ROCKET




BOAT


CASTLE

length width
width first

GRAPH PAPER ON NEXT PAGE

? ? ? ? ?

whatever you want

Copyright 1974, A. B. Cannara and S. A. Weyer.

Write a procedure called SECOND which outputs the second letter of an input word (or second word of an input sentence).

Write a procedure called THIRD which outputs the third letter of an input word (or third word of an input sentence).

Write a procedure called SWITCH13 that takes its input word, interchanges the first and third letters, and outputs the resulting word. Your procedure should change

these words	into these words
FIRST	RIFST
PICTURE	CIPTURE
CALENDAR	LACENDAR
GORILLA	ROGILLA
EVEN	EVEN

Does SWITCH13 undo itself? ....

When you type	what do you get?
P SWITCH13 SWITCH13 "HOUSE"	....
P SWITCH13 SWITCH13 "123"	....
P SWITCH13 SWIICH13 "WATERMELON"	....

Sometimes there are pairs of functions, each of which undoes the work of the other. We say that they are "inverses" of each other. DOUBLE and UNDOUBLE were not really inverses for all numbers; they were inverses for only .... numbers. What is the inverse function for SWITCH13? .... Try SWITCH13 with some inputs which have less than three letters and write what happened ....

Write a procedure called AGAIN that doubles its input word (its input is a word), and outputs the resulting word.

when you type this	you should get
P AGAIN "DOG"	DOGDOG
P AGAIN AGAIN "ALDO"	ALDOALDOALDOALDO
P AGAIN W "BL" AND "ACK"	BLACKBLACK
P AGAIN (EMPTY)	
P AGAIN 12345	1234512345

Is AGAIN a function? ....

Write a procedure that takes its input word, moves the first letter to the end of the word, and outputs the resulting word. Your procedure should change

these words	into each of these words.
LAMP	AMPL
JABBERWOCKY	ABBERWOCKYJ
QWERTY	WERTYQ
12345	23451

Write a procedure called FUNNYADD that takes its two inputs (they are numbers), adds the first digit of each input number and outputs the resulting number.

When you type	you should get
P FUNNYADD 87 15	9
P FUNNYADD 1 99999	10
P FUNNYADD 18 FUNNYADD 77 200	10
P FUNNYADD FUNNYADD 38 75 FUNNYADD 25 66	9

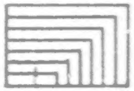
Copyright 1974, A. B. Cannara and S. A. Weyer.

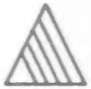
Write a recursive turtle procedure (called BEND or some other name) that draws a line, turns, draws a line, turns, etc. You should give it two inputs: the first input is the distance the turtle moves; the second input is the angle to turn. By giving BEND different inputs, see if you can get it to draw a hexagon, an octagon, and a circle.

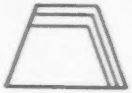
A procedure can tell different inputs to its brother on the right. Are the brothers' knowledge clouds the same? ....Nope

Write a recursive turtle procedure which spirals inward. Modify your BEND procedure so that when it calls itself recursively, it will turn more than it just turned (this amount of increase or decrease could also be an input).

Write a turtle procedure to draw some "nested" figure. First, write a procedure for the figure (design your own, or use something like a square, triangle, or hexagon) which has one or more inputs, for example, length of side or turning angle. Then write a recursive procedure which calls upon the figure with inputs to make the figure become smaller or larger.







Write a recursive procedure (called DIAGDASH) that types a diagonal dashed line (which looks like -) on your typewriter. Give your procedure an input so that you can change the "dashing" character from "-" to "*" or "Z". Logo already knows about :LINE FEED: which when TYPEd, moves to the next line on the typewriter without going to the beginning of the line. (The LINEFEED command is the same as TYPE :LINE FEED:)

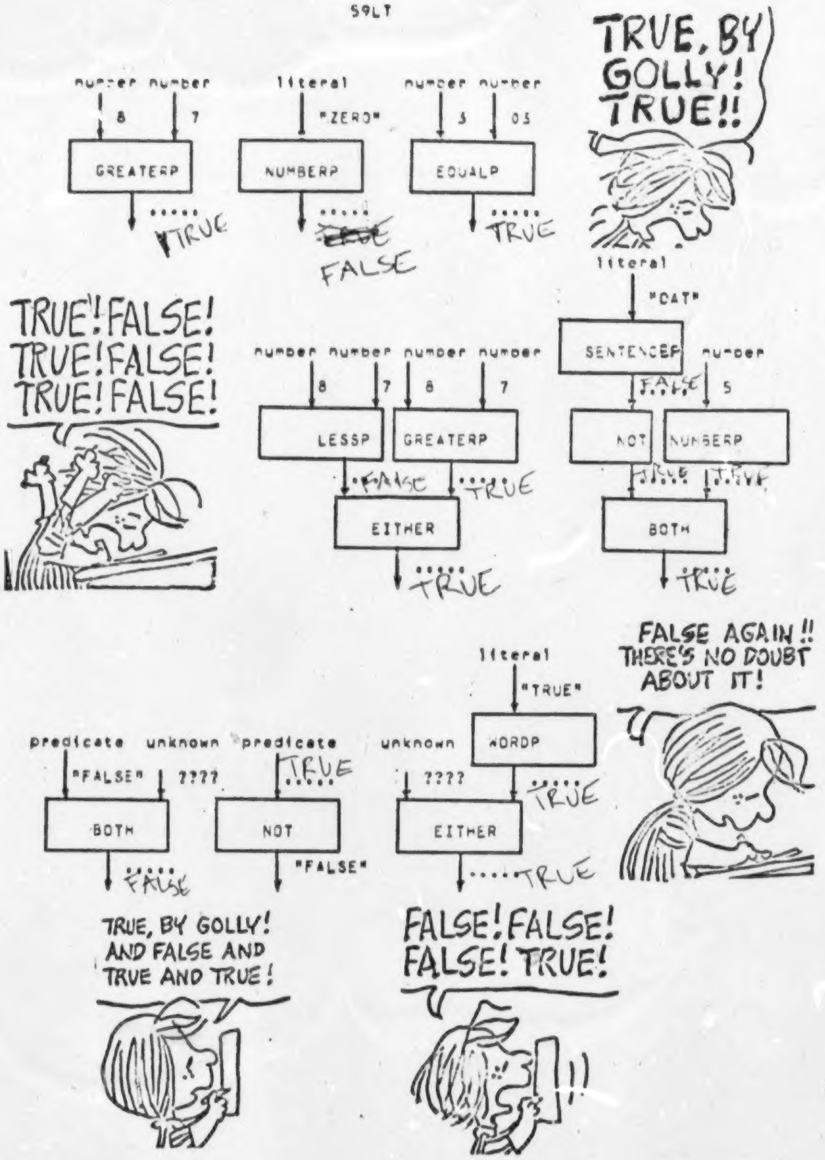
A procedure can tell different inputs to its brother on the right. Are the brothers' knowledge clouds the same? .....

Write a recursive procedure which does "ripple" printing. It prints its input, and then calls itself with an input which has the first letter moved to the end of the same word (or the first word moved to the end of the same sentence). This problem should look familiar (see pages 47LT and 48LT).

RIPPLE "BANANAS" should give you

```
BANANAS
ANANASB
NANASBA
ANASBAN
NASBANA
etc.
```

Copyright 1974, A. B. Cannara
and S. A. Weyer.



Copyright 1974, A. B. Cannara and S. A. Weyer.

Part 8

Computers are able to make decisions about what to do next depending on what has happened already. For example, putting fifteen cents into a candy machine and pulling the M&M's knob does not guarantee that you will get M&M's. If the machine is out of M&M's, then you can select another item. If it is out of candy entirely, it will return your money. You make comparisons and decisions every day: is a six-pack of the uncola better than 2 quart bottles? should you go swimming or play tennis this afternoon? does Scope taste better than Listerine? is Mt. Everest taller than Mt. Shasta? is the bathroom empty yet?

Logo asks other kinds of questions like: is 6 greater than 7? is "GEORGE" a word? is "XYZ" a number? To any of these Logo gives the answer "TRUE" or "FALSE".

When Logo obeys the TEST command with an input whose value is "TRUE", you can imagine that Logo turns on a light that can be seen everywhere in your procedure. When the value of its input is "FALSE", TEST turns off the light.

when Logo obeys this command	and the light from TEST is on ("TRUE")	and the light from TEST is off ("FALSE")
IFTRUE	Logo obeys the command following IFTRUE	Logo ignores the command following IFTRUE
IFFALSE	Logo ignores the command following IFFALSE	Logo obeys the command following IFTRUE

63L

The only living creatures on the planet Binar resemble zeros and ones. A colony of these life forms from Binar (who call themselves "bigits") look deceptively like a sequence of 0's and 1's, for example, 100011010011. Your mission (should you decide to accept it) is to write a recursive procedure which will help the colony to reproduce for several generations. Your procedure should print what the current colony looks like. Then if there are less than 3 bigits remaining in the colony (the COUNT command may be handy here), it should die, so your procedure can print "EXTINCTION!!" (or another appropriate message) and stop. Otherwise, it should follow these mating rules:

a. If the first bigit is a 1, then the first three bigits die and four new bigits (1011) are born at the right end of the colony. For example, if the colony is 10001, then the next generation is 011011.

b. If the first bigit is a 0, then the first three bigits die and two new bigits (00) are born at the right end of the colony. For example, if the colony is 011011, then the next generation is 01100.

c. If the first bigit is not a 0 or a 1, then your procedure should use the EXIT command to cause a Logo error message, for example, IFFALSE EXIT "ALIEN LIFE FORM"

an example of how your procedure might look,

```
*BINAR 10001
10001
011011
01100
0000
000
00
EXTINCTION!
```

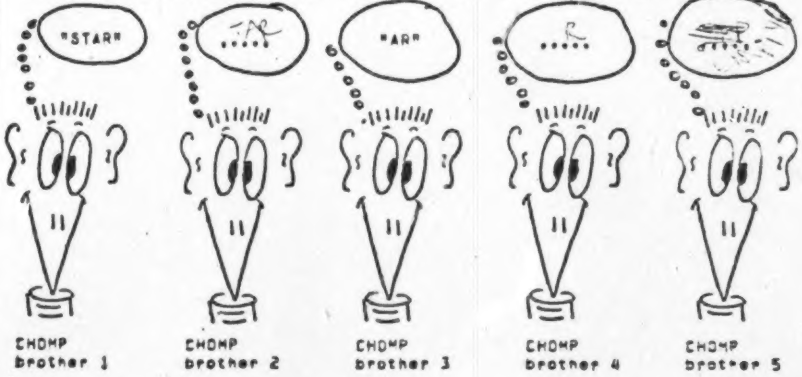
61LT

You've probably wondered: "how can I get a recursive procedure to stop by itself without having to type control-G?" With the TEST command, you can find out the answers to certain questions (like is my input equal to zero, or is the value of my sentence empty?) and then tell Logo (with IFTRUE or IFFALSE) to STOP the procedure or OUTPUT a value (which also stops a procedure).

Type and TRACE the following procedure. If you've forgotten how to TRACE a procedure, ask a tutor.

```
TO CHOMP :WORD:
10 TEST EMPTYP :WORD:
20 IFTRUE STOP
30 PRINT :WORD:
40 CHOMP BUTFIRST OF :WORD:
50 PRINT :WORD:
END
CHOMP "STAR"
```

Fill in the missing knowledge clouds for the CHOMP brothers.



Copyright 1974, A. B. Cannara and S. A. Weyer.

Write a recursive procedure called FIND which finds and outputs any letter in a word. The first input should be the letter's position from the beginning (1 is the first etc.) of the word. The second input should be the word you want to look in. If the word is empty (EMPTYP would output "TRUE"), then FIND outputs the :EMPTY: value. This means that the letter position is (less than, greater than, the same as)? the length of the word. If the letter you want is the first letter (the letter position is 1), then your procedure outputs First letter.............. Otherwise, your procedure should (fill in the blanks!)

OUTPUT FIND	....	....
	(choose one)	(choose one)
(Remember!	SUM :POS: 1	BF :WORD:
You can TRACE	:POS:	F :WORD:
a procedure to	"POS"	BL :WORD:
see what its	BF :POS:	F "WORD"
inputs and	DIFF :POS: 1	:WORD:
output are.)		

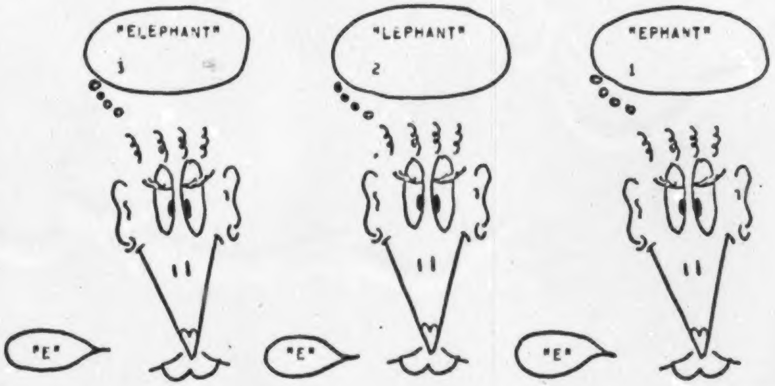
Type:

```
MAKE "ALPHABET" "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
MAKE "DISEASE" "PNEUMONOULTRAMICROSCOPICSILICOVULCANOCONEOSIS"
MAKE "DIGITS" "ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE TEN"
P FIND 29 "SUPERCALIFRAGILISTICEXPIALADOCIOUS"
P FIND 13 :ALPHABET:                ....M.....
P FIND 40 :DISEASE:                 ...N......
P FIND 8 :DIGITS:                   ..Eight....

P COUNT "DRACULA"                   ..45.....
P COUNT :DISEASE:                   ....7.....
P COUNT "THREE WORDS"               ....2....
```

(The COUNT command outputs how many letters are in a word or how many words are in a sentence.)

Remember when you wrote procedures to find the SECOND and THIRD letters in a word (part 7)? Imagine what it would be like if you had to find the tenth or fiftieth letter! How do you find the first letter in a word or first word in a sentence? ...USE THE "F" command... The inputs to CHOMP (part 8) were always losing their first letters and getting shorter. Eventually the letter you wanted would show up at the beginning of the word. Do you think that recursive procedures can output values? ..Yes..... The value that a procedure outputs is contained in a talk or output "bubble". Here are the FIND sisters who can find the letter at any position in a word (or the word at any position in a sentence) and then report back what that letter (or word) is. For example, they can tell you the second, or fiftieth, or "nth" letter (where n is any number). Do the FIND talk bubbles below contain the same output? ..Yes.......

FIND 3 "ELEPHANT"



195

Modify EVENP so that it uses MEMBERP. Assume that you can't use REMAINDER anymore. How else can you tell if a number is even? ....

Write a procedure HASNUMBERP which outputs "TRUE" if its input word has any digits in it, and "FALSE" if its input has no digits. (Hint: use NUMBERP or MEMBERP.)

```
P HASNUMBERP "THREE"
FALSE

P HASNUMBERP "DOG3"
TRUE
```

Write a procedure called HAMBURGERP which outputs "TRUE" if its input is the name of a hamburger, and "FALSE" if its input is not a hamburger. (Hint: use MEMBERP.)

```
P HAMBURGERP "BIGMAC"
TRUE

P HAMBURGERP "BONUSJACK"
TRUE

P HAMBURGERP "PIZZA"
FALSE
```

Rewrite VOWELP so that it uses MEMBERP. Your procedure outputs "TRUE" if its input is a vowel, and "FALSE" otherwise. What is the set of letters which are vowels? ....

Fix PIG (page 60L and 60T) so that all leading consonants are moved to the right end of the word before adding "AY". (Hint: if both the first and second letters are consonants, you can move the first letter, and then use the PIG procedure recursively). If the input word consists of all consonants, what happens? ....

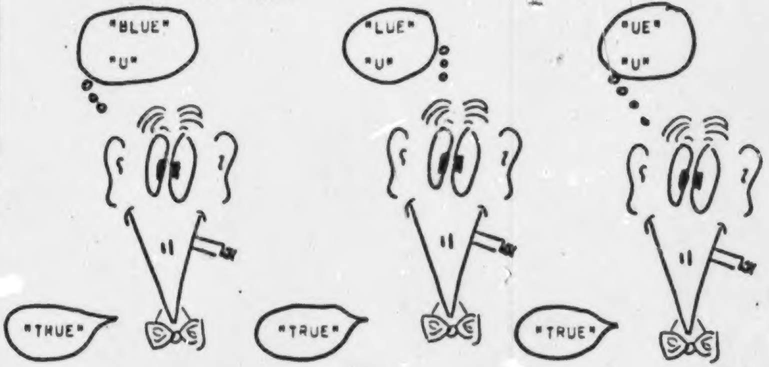
Write a recursive procedure (called MEMBERP or ISIN) which has two inputs and outputs "TRUE" if the first input is included in (or is a member of the set which is) the second input. For example,

```
        TO MEMBERP :ELEMENT: :SET:

        or

        TO ISIN :THIS: :THESE:
```

```
P MEMBERP "7" "ABCDEFH"
FALSE

P MEMBERP "APPLE" "APRICOT ORANGE APPLE CHERRY"
TRUE
```

Here are the MEMBERP brothers' knowledge clouds and talk bubbles.

```
        MEMBERP "U" "BLUE"
```



Copyright 1974, A. B. Cannara and S. A. Weyer.

In the "modern" Roman system, a number is also a sequence of M's, D'S, C's, L's, X's, V's, and I's. The symbols have to appear more or less in that order and the value of a number is obtained just as in the "old" Roman system except when a symbol C, X, or I precedes a symbol of higher value. In that case, the value of that symbol C, X, or I is taken to be negative. For example,

decimal	"old" Roman	"modern" Roman
4	IIII	IV
9	VIIII	IX
91	LXXXXI	XCI

Write a procedure to convert both "old" and "modern" Roman numerals into decimal numbers. For example,

```
P UNROMAN "XXV"
25

P UNROMAN "CXIV"
114
```

Write a procedure to convert decimal numbers into "old" (easy) or "modern" (harder) Roman numerals.

```
P ROMAN 37                     P ROMAN 37
XXXVII                         XXXVII

P ROMAN 199                    P ROMAN 199
CLXXXXVIIII                    CIC
```

Write a procedure which outputs the sum of all of the digits in an input word (or numbers in an input sentence).

```
P SADD "5 6 7 9"
27

P SADD "993"
21
```

Recent discoveries by Profesora Ursula de Logo on the planet Wazzenazz 3 indicate that the Ursinoid inhabitants (they look like bears) do not know how to divide by three in the same way that humans do. Instead, the ursinoids claim that if the sum of all the digits in a number is divisible by three, then the number itself is divisible by 3 (For example, 33 is divisible by 3 because 3 + 3 is divisible by 3). If the digit sum is not divisible by three, then the number is not divisible by three. Please answer "TRUE" or "FALSE" in the blanks below:

	divisible by 3?		divisible by 3?
37	....	57	....
3 + 7 = 10	....	5 + 7 = 12	....
1 + 0 = 1	....	1 + 2 = 3	....
105	....	999	....
1 + 0 + 5 = 6	....	9 + 9 + 9 = 27	....
		2 + 7 = 9	....

You can help Ursula test the Ursinoid theory by writing two procedures which output "TRUE" if a number is divisible by 3 and outputs "FALSE" if not. The first procedure can use the regular Earth method, which is the .... command. The second procedure should use the Wazzenazz method (no QUOTIENT, DIVISION, or REMAINDER commands allowed!) (Hint: make this procedure recursive, use SADD to add the digits, and

Copyright 1974, A. B. Cannara and S. A. Weyer.

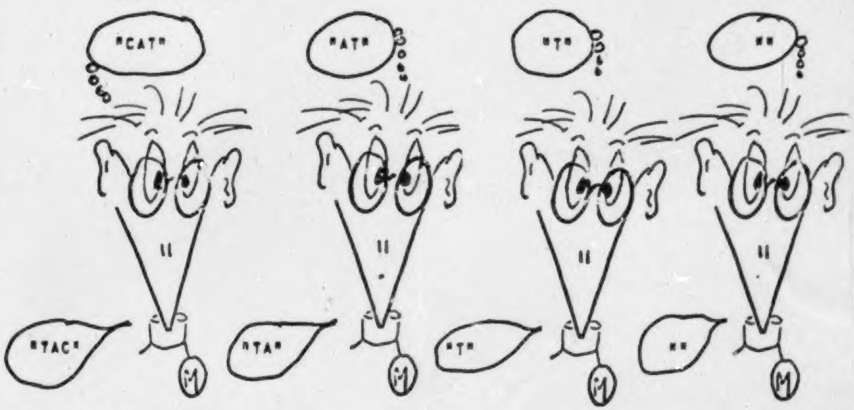
Part 10

In our efforts to communicate with the creatures on the planet Mirror, we have discovered that the Mirrorians have to read our messages backwards and we have to read their messages backwards. To aid in intergalactic friendship, write a procedure (called REVERSE) to reverse a word. For example,

```
        P REVERSE "ELEPHANT"                P REVERSE "BACKWARDS"
        TNAHPELE                            SDRAWKCAB

        P REVERSE "SUMMER"                  P REVERSE "ESREVER"
        REMMUS                              REVERSE
```

As usual, there are several ways to write the REVERSE procedure since you can start at either end of the word. Here is one family of REVERSE brothers and their knowledge clouds and talk bubbles.

REVERSE "CAT"



Here is a numbering system based on U.S. coins:

Symbol	coin name	value in cents
H	half dollar	50
Q	quarter	25
D	dime	10
N	nickel	5
P	penny	1

Write a procedure that converts a group of coins into the correct number of cents.

```
P MONEY "HQD"             P MONEY "NPPP"
85                        8
```

Write a procedure that converts cents into a group of coins which represent the same amount of money. Since there are sometimes several groups of coins that are worth the same amount of money, use as few coins as possible. For example, 99 cents is better represented by HQDDPPPP than by PPPPPPPPPPPPP . . . P (99 P's).

```
P CHANGE 107              P CHANGE 46
HHNPP                     QDDP
```

Use English as a numbering system and write procedures to translate words into numbers and numbers into words.

```
        easy                                harder
        ----                                ------

        P ENGLISH 123                       P ENGLISH 123
        ONE TWO THREE                       ONE HUNDRED TWENTY THREE

        P NUMBER "ONE FIVE SIX TWO"         P NUMBER "ONE THOUSAND FIVE
        1562                                 HUNDRED AND SIXTY TWO"
                                            1562
```

Copyright 1974, A. B. Cannara and S. A. Weyer.

The birdlike creatures on Aviary 2 speak to each other in palindromes. Palindromes are sentences that read the same backwards or forwards. Here are some examples donated by that dangerous duo, Batman and Robin (who consider themselves distant relatives of the Aviarians).

A MAN A PLAN A CANAL PANAMA

ABLE WAS I ERE I SAW ELBA (said by Napoleon?)

EIN NEGER MIT GAZELLE ZAGT IM REGEN NIE (German)

Batman and Robin are planning a trip to Aviary 2 but could find no foreign language books which can tell them whether or not certain phrases are palindromes. Write a Logo procedure which will tell them whether or not a given sentence is a palindrome. (Hint! use your CRUNCH and REVERSE procedures.)

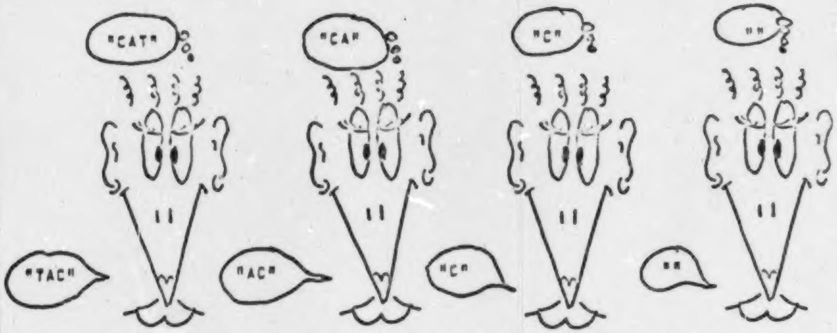
```
P PALINDROME "PALINDROME"
FALSE

P PALINDROME "MADAM IM ADAM"
TRUE
```

"RATS A STAR"

can you think of any other palindromes? ....

Here is a family of REVERSE sisters.

REVERSE "CAT"



In both of these REVERSE procedures, if the input is empty, your REVERSE procedure outputs .... If the input is not empty, your REVERSE removes a letter from one end of the input and glues it to the rest of the reversed word. Try it, TRACE it, and talk to a tutor if you get stuck.

Since the Mirrorians like to speak in sentences, write a procedure which reverses an entire sentence.

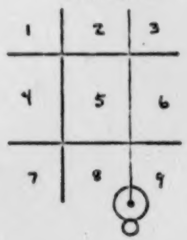
```
P REVS "PEANUT BUTTER BUT NO JELLY"
YLLEJ ON TUB RETTUB TUNAEP

P REVS "REVS NEEDS A MEANING"
GNINAEM A SDEEN SVER
```

You must reverse both the positions of the words in the sentence and the words themselves. For example, "JELLY NO BUT BUTTER PEANUT" is one way to reverse the sentence "PEANUT BUTTER BUT NO JELLY" but not the way the Mirrorians talk.

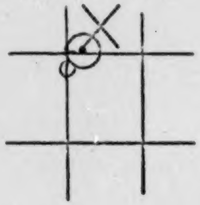
Copyright 1974, A. B. Cannara and S. A. Weyer.

There is a game invented by sponges on the planet Lombogo which is called Turtletoes. We have been given drawings of the board and various markers (called "X" and "O") which are placed on the board alternately by two players. The rules look exactly like the Earth game of Tic-tac-toe. Write a procedures to draw the Turtletoes board and another procedure to place a marker in a square.

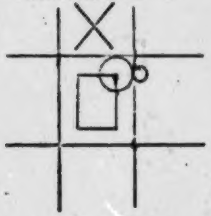
DRAWBOARD



PUTMARKER "X" 2



PUTMARKER "O" 5



(hint: you should always move the turtle back to the same resting point. You can make some names which have as their values, a sentence which gives directions to the turtle about how to get to a square and then use your DRAW procedure. For example,

```
MAKE "SQ1" "U F100 L90 F300"
MAKE "SQ2" "U F200 L90 F300"
```

(if :WHERETO: contains the number of the square you want) then

```
DRAW THING OF WORD OF "SQ" AND :WHERETO:
```

moves the turtle to the square where you want to put an "X" or "O".

You can return the turtle to its resting point by using DRAW with another set of names (like RT1, RT2, etc.) or a picture reversing procedure (which you can't quite do yet) or by using "home" as the resting point.

Write a turtle procedure which draws a picture from a description given by its input. For example,

```
DRAW "D F100 R90 F100 R90 F100 R90 F100 R90 U"
```

The DRAW procedure translates each of

these commands (or make up your own!)	into these turtle commands.
D	PENDOWN
U	PENUP
F followed by a number	FRONT that number
R followed by a number	RIGHT that number
etc.	etc.

So in the example, DRAW "D F100 R90 F100 R90 F100 R90 F100 R90 U" would draw a .... Since you are inventing a new language, you may want to give your own error messages.

```
DRAW "XYZ 200"
XYZ IS AN UNKNOWN COMMAND

DRAW "100 200"
100 IS NOT A COMMAND BY ITSELF

DRAW "FABC U"
ABC CANNOT BE AN INPUT FOR FRONT
```

(although you may want to fix DRAW so that it uses :ABC:)

Since you are representing a picture as a sentence, you could "reverse" a picture by changing the positions of the commands (words) in the picture (sentence). In a little while, you will learn how to finish reversing the picture by replacing the letter "B" by "F", "L" by "R" etc.

Copyright 1974, A. B. Cannara and S. A. Weyer.

Write a procedure which replaces one letter by another letter (or word) in a word.

```
TO REPLACE :THIS: :BY,THIS: :IN,THIS:
; you fill in the rest
END

P REPLACE "E" "O" "FEED"
FOOD

P REPLACE "A" "" "ABRACADABRA"
BRCDBR
```

Your REPLACE procedure replaces the first input by the second input whenever it sees it in the third input. If the third input (the word you are making replacements in) is empty, then your REPLACE should OUTPUT .... when you find a letter to replace, REPLACE should

```
OUTPUT WORD OF   ...........
                 (choose one -- several may be correct)
                 :THIS:
                 :BY,THIS:
                 REPLACE :THIS:        :BY,THIS:      BL :IN,THIS:
                 REPLACE  BL :THIS:     BF :BY,THIS:  L :IN,THIS:
                 F :IN,THIS:
                 DIFF :THIS: 1

       AND       ...........
                 (choose one -- several may be correct)
                 REPLACE :THIS:        :BY,THIS:      :IN,THIS:
                 REPLACE :THIS:        :BY,THIS:      BF  :IN,THIS:
                 :THIS:
                 REPLACE  BF :THIS:    BF :BY,THIS:   BF  :IN,THIS:
                 :BY,THIS:
                L :IN,THIS:
```

If the letter in the word you are looking at is not one you want to replace, then you should

```
OUTPUT WORD OF .................... AND ....................
               (choose one from         (choose one from
               the list on the          the list on the
               last page)               last page)
```

Logo is more powerful than a Captain Midnight decoder wheel for secret messages. You can not only reverse words, and move letters around, but also replace letters in words by other letters. Write a procedure to translate a word from one code to another.

```
MAKE "ABET" "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
MAKE "COD1" "QWERTYUIOPASDFGHJKLZXCVBNM"

TO CODE :WRD: :ALPHABET: :CODE:
; you figure out the rest
END

P CODE "WATERGATE"  :ABET: :COD1:
VQZTKUQZT

P CODE "VQZTKUQZT"  :COD1: :ABET:
WATERGATE
```

The first input to your CODE procedure should be the word you want encoded or decoded. The second and third inputs are the letters of the alphabet and their corresponding codes. For example, "A" is to be coded as "Q", "B" as "W" etc. Your CODE procedure would then

Copyright 1974, A. B. Cannara and S. A. Weyer.

If you've thought of a problem, game, picture, or project you would like to work on now, talk to a tutor. If not, here are more problems to try that might give you some ideas,

Draw the reverse of a picture (which is represented by a sentence of commands) by reversing word positions in the command sentence and by replacing FRONT by BACK (F by B), BACK by FRONT (B by F) etc. (hint: use your CODE procedure). For example,

```
P REVERSEPIC "D F100 R90 B200 L10 U"
D R10 F200 L90 B100 U
```

Modify your DRAW procedure to make a new command language for the turtle. When DRAW sees a number followed by something contained in parentheses, it does what is in parentheses that number of times. For example, DRAW "4(F100 R90)" will go FRONT 100 and RIGHT 90 four times. Be careful about "nesting" of parentheses, for example: DRAW "4(F100 2(R45))". (Hint: you should be able to use your DRAW procedure to do the drawing and your scan procedures to find and count parentheses.

Use the turtle to draw a clock face and the hour and minute hands in the correct TIME position (hint: use your procedures from page 84LT to output the hour and minutes from TIME). Put in a "time lapse" input (normally this is 60 seconds/minute) which you can use to speed up or slow down your clock.

Write a procedure which draws a road map between cities, for example, MAP "LOSANGLES SACRAMENTO DENVER CHICAGO".

Write two procedures called SCANF (or SCANFIRST) and SCANBF (or SCANBUTFIRST) which have two inputs. SCANF outputs everything up to the occurrence of the first input in the second input. For example,

```
P SCANF "$" "ABC$DEF$GHI"
ABC

P SCANF "V" "AARDVARK"
AARD

P SCANF "%" "123456789"
123456789                      (it didn't find %)
```

SCANBF outputs everything after the occurrence of the first input in the second input. For example,

```
P SCANBF "$" "ABC$DEF$GHI"
DEF$GHI

P SCANBF "$" SCANBF "$" "ABC$DEF$GHI"
GHI

P SCANF "," SCANBF "$" "QWERTYUIOP$123,7"
123

P SCANBF "," "37,314"
314
```

The Timex company is considering a smart watch (run by Logo) which does not tell time as "9:14 PM" or "11:02 AM". The watch will will say (print) things like

```
FOURTEEN MINUTES PAST NINE O'CLOCK                  instead of 9:14
58 MINUTES BEFORE TWELVE O'CLOCK IN THE MORNING     instead of 11:02
THE BIG HAND IS ON THE 3 AND
THE LITTLE HAND IS ON THE TWELVE                    instead of 12:15
```

Write a procedure to run this watch. (hint: use your SCANF and SCANBF procedures to get the hour and minutes from TIME by looking for ":" and :BLANK: (before AM and PM)). If this problem excites you, you might want to write another procedure which knows about dates on calendar watches (hint: scan for "/" in DATE).

Copyright 1974, A. B. Cannara and S. A. Weyer.

Appendix 5: Sample Simper Curriculum

This appendix contains portions of the Simper curriculum developed for the experiment discussed in this report. Many of the excerpts shown here are referenced by discussions in the text, particularly in Sections 4.2 and 6.1. The following table indexes the curriculum by part number, curriculum page, and page of this appendix. The text is copyrighted, but may be used for noncommercial purposes.

Part	Curriculum Page	Page
1	see Logo index	172
2	6,12	196
3	13,15,17,19	197-198
4	22,24	199
5	27,30,35,37	200-201
6	40,45	202
7	46-48,52,53	203-205
8	57,60	205-206
9	61,62	206-207
10	67-69,72,73	207-209
11	77-80	210-211
12	82,85-87,89	212-214
13	91,94,97	214-215

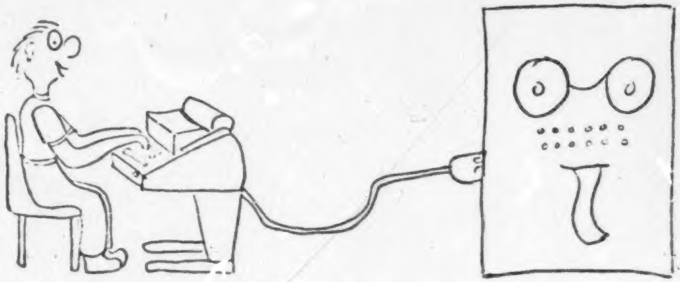
Here's the meaning of all the control commands in Simper's language:

command	means
RUBOUT or control-A	erase the last character on my line
control-W	erase the last word on my line
control-U or control-X	erase my whole line
control-R	type what my line really looks like
LINEFEED	continue my line
RETURN or ENTER	I'm finished typing my line
control-G	stop what you're doing
control-Z	log me out

Do you understand each one?

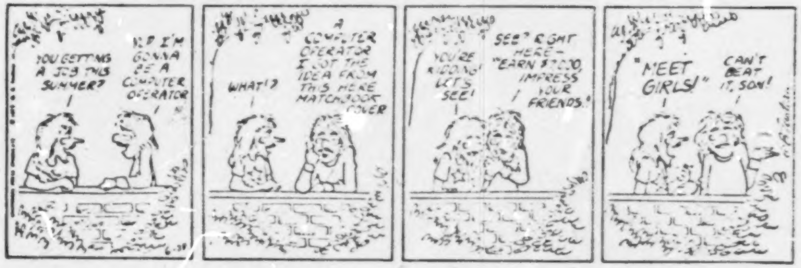
YES → Use them whenever you want to. The moral of this is:

NO → ask one of the tutors to help you → Here's the meaning of all the control commands in Simper's language:

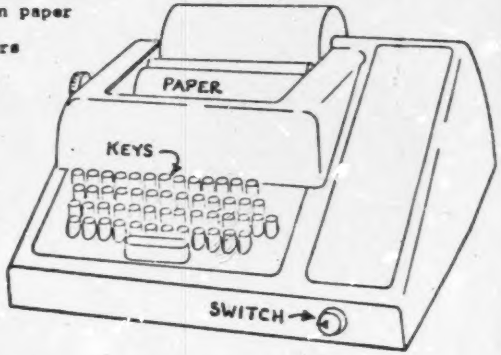


YOU and the TYPEWRITER CONTROL the Simper COMPUTER

Computers and computer languages are designed by people and, since people don't yet know how to make a computer understand English or Chinese, we have to learn the simple languages that people have succeeded in making computers understand. When you've learned how to program a computer, you can try to make up your own computer languages.


YOU GETTING A JOB THIS SUMMER?
YEP I'M GONNA BE A COMPUTER OPERATOR
WHAT'?
A COMPUTER OPERATOR I GOT THE IDEA FROM THIS HERE MATCHBOOK COVER
YOU'RE KIDDING! LET'S SEE!
SEE? RIGHT HERE— "EARN $[illegible], IMPRESS YOUR FRIENDS."
"MEET GIRLS!"
CAN'T BEAT IT, SON!

One thing about the way computers are made, they pay attention to what we tell them. And the most common way of telling them things is by typing on the electric typewriters that most of them have. That's where they give their attention. You can see two kinds of typewriter in the pictures on the previous page. The lefthand one prints on paper that a person can save, while the other prints on a television screen. Some people prefer TV typewriters because a computer can print whatever it has to say faster on a TV screen than on paper, but what it prints on paper doesn't disappear, it's yours forever. Here's a drawing (made by a human) of the kind of typewriter you'll be using:



Copyright 1974, A. B. Cannara and S. A. Weyer.

street and look inside each house. Simper's memory is just like that. There are 250 places, we call them "locations", in Simper's memory and each one can hold a number with as many as ten digits in it. So far, you've put numbers into a few locations by simply typing a literal when Simper put you at a new location.

LOCATION	[___]	[___]	[___]	[___]	...	[___]
ADDRESS	001	002	003	004	...	250

So the number to the left of the ":" Simper types is the address of the location you can put something into next. It's like standing in front of the house with that address. You can open the door and put something inside by typing a literal and the RETURN key. Suppose all the houses on the street are new and nothing is in them yet. That's what Simper's memory is like when you first log on. Each memory location contains the number 0. Remember, anytime you want to see what's in Simper's memory, just use the LIST command.

Now that you can make Simper remember things, something any computer must be able to do, you are ready to make it forget everything.

→ Type SCRATCH and the RETURN key. Now use LIST to see what Simper remembers.

Did Simper remember your old numbers?

YES ↓ It shouldn't have!

NO ↓

Good, the SCRATCH command gives Simper permanent amnesia. All the locations (houses) are empty again, as if you had just logged on.

Part 3

So far, you've learned some helpful commands for typing and changing lines before Simper reads them. They are called "editing commands", because they let you do what the editor of a newspaper does, he changes stories going into the paper before the public gets to read it. Now we're really going to start talking Simper's language.

Simper, like most computer languages, understands several kinds of words that you can type. You've already learned to use some of the kind called "command". Another kind is something else you've probably seen already. It's called a "literal". When you talk to Simper and you mention a number, any number, that number is a literal. One thing about a literal, it's not hiding anything. You'll always know what it means by just looking at it.

See! Nothing up my sleeves

Luigi Literal

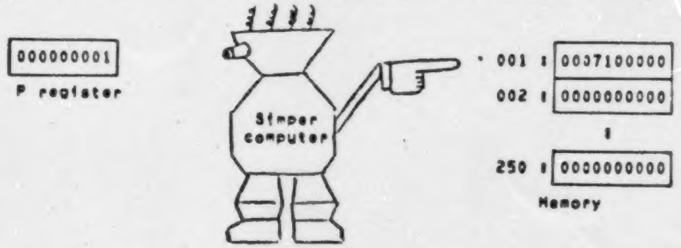
In English, people often use quote marks (") to surround words that are part of a literal. Look at this sentence:

The cookie monster said: "Give me cookies!".

The cookie monster actually said the words between the quotes. The words between the quotes make a literal in English. Simper, however, does NOT use quotes to mark literals. Except in one case, numbers are its only literals.

Copyright 1974, A. B. Cannara and S. A. Weyer.

The RUN command tells the Simper computer that you have put a program (a list of commands) into its memory and you want it to obey them. "How does the computer obey my instructions?", you ask. When you give the command RUN, Simper puts the number 1 into the P register and tells the Simper computer to start obeying instructions in its memory (that's all the computer knows how to do). The Simper computer comes to life. It looks at the P register to see what's in it:



It uses the number in the P register as an address and looks in its memory for the location which has that number for its address. This is similar to what we do when we have a friend's address on a piece of paper and we use it to find the right house.

As soon as the Simper computer finds the location the P register says to find, it makes the number in the P register 1 bigger than it was before (you'll see why it does this soon). Now the computer reads the number in the memory location it has found and obeys that as an instruction (you probably remember that most computers really only understand a language of numbers).

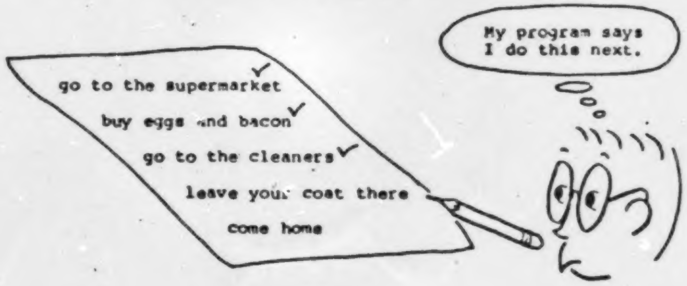
In the picture above, is the instruction in location 1 a HALT ?

YES ↓

NO ↓

Type HALT and the ENTER key.

Simper's registers are yours to use for calculations when you write programs. "What is a program?", you ask. A program is a list of instructions that some human or machine can read and obey.



You've already been programming by using the editing, LIST, DUMP and SCRATCH commands. However, Simper obeys those commands immediately without waiting for you to make a list of them. There are, in fact, some commands, called "instructions", which Simper will let you put in a list for it to obey later. When we make a list for another person, we usually write it on paper. For Simper, lists must be written into its memory. You already know how to type numbers into Simper's memory and then LIST and DUMP them. Now you'll see how to type instructions into memory and use LIST to see them.

Use SCRATCH to clear Simper's memory. Now type HALT and an ENTER.

What number did Simper print next to your HALT? 710000

That number is what Simper actually has in its memory. It's how Simper translates HALT. The Simper computer, like most computers, really only understands numbers. Simper reads what you type and, if it is an instruction for the Simper computer, it puts a number

Copyright 1974, A. B. Cannara and S. A. Weyer.

Is the address of the location you are at now 2?

YES → Ok, type PUT P and a space and the address you are at now (the number to the left of the exclamation mark).

NO → It should be! Get some help.

What do you think the value in the P register will be after Simper obeys this instruction? ______

What location do you think Simper will look at next after it obeys this instruction? ______

Believe it or not, you have just written a program that will go forever!

```
001 !PUT A 73
002 !PUT P 2
003 !HALT
```

FOREVER !?!!!!

RUN it, but first remember that the control G command stops Simper from doing something no matter what that something is. Now tell Simper to obey your "forever" program by typing RUN and an ENTER.

Please show what Simper is typing!

Is the value in the P register changing now?

YES → Ask for help.

NO ↓

Is any register's value changing now?

YES → Ask for help.

NO ↓

Remember that Simper's memory is like a street with 250 houses on it. Imagine someone told you to walk down that street and stop at each house to collect money for UNICEF. Which of these two collection trips would you rather take?

HOUSE ADDRESSES

START ↗ 15 → 230 → 23 → 1 → 87 → 176 → ... TRIP 1

START ↘ 1 → 2 → 3 → 4 → 5 → 6 → ... TRIP 2

If you chose trip 1, you'd probably be very tired by the time you had visited every house and you might have forgotten some. Trip 2 is shorter and easier to know how to follow. Most computers don't get tired, but they aren't very smart. So trip 2 is easier for Simper to follow, because all it must do is add 1 to the address of the location it is looking at now in order to get the address of the location to look at next. The P register is where Simper holds this bit of information while it is obeying the instruction it is looking at. Give the SCRATCH command to erase memory and type in this new program (you type what's underlined and end each line with ENTER or RETURN as usual):

```
001 !PUT A 73
     --------
002 !PUT B 66
     --------
003 !HALT
     ----
```

Now run your new program by typing RUN and the ENTER key. Observe the P register carefully as each instruction is obeyed.

Copyright 1974, A. B. Cannara and S. A. Weyer.

Since Simper has three parts (interpreter, assembler, computer), you are really learning to talk three languages. The first is the interpreter's "command language" which consists of the commands like ENTER, RUBOUT, the control characters, RUN, LIST, and more commands that you will be learning soon. The second language is made of all the possible instructions like SUB A 23, HALT, and many more you will learn. This language is called Simper's "assembly language", because the assembler takes each part of an assembly language instruction, translates it into a number and assembles these numbers, from left to right, to make one number that goes into the Simper computer's memory!

```
    PUT            B              45
     \\             \\             \\
     11             1             0045   →    1110045
```

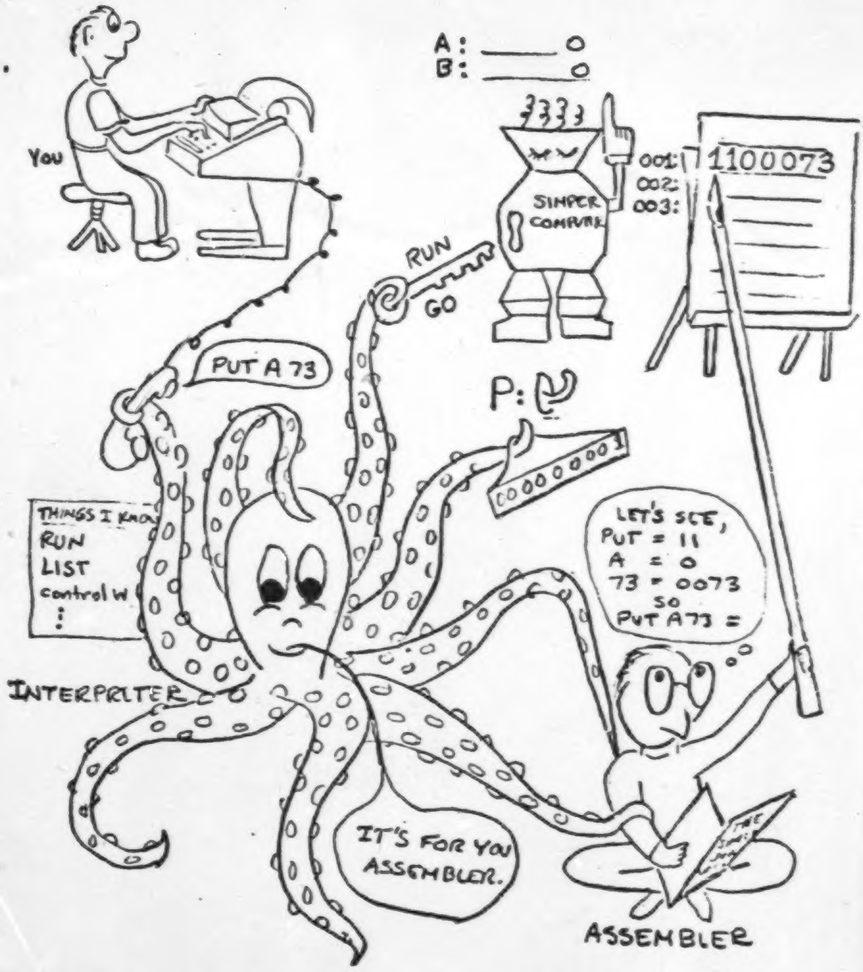
Most assembly language instructions have three parts called "fields". From left to right they are the operation field (PUT), the register field (B), and the address field (45).

The third language you're learning is Simper's "machine language". It is simply all the numbers, like 1110045, that the Simper computer understands. Remember that, when you say RUN, the Simper computer can obey only those numbers in its memory that it understands. It will complain if the P register ever tells it to look at some location that contains a number it does not know how to obey. Use SCRATCH to erase Simper's memory and then type these literals into memory!

```
001 !0
002 !10
003 !1000000
004 !6210000
005 !7100000
```

Part 5

By now, you may have a feeling that Simper has a split personality. You're right. It really has three parts: the interpreter, the assembler and the computer. This picture shows it all!



Copyright 1974, A. B. Cannara and S. A. Weyer.

Would the program work correctly now (stop only if you type 0)?

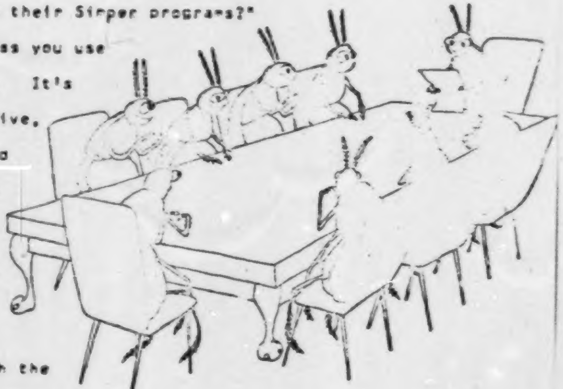
NO

YES

Check what happened with the first bug, because this is a new one of the same species.

What's the bug now It's still testing the wrong register...?

An important thing to remember about bugs is that they might look like mistakes at first, but later you will learn to see their "ant"-ics as funny, or even useful. Remember, the Countess of Lovelace in Part 1? She discovered bugs in Babbage's machine which made it work better than he dreamed. Unfortunately, that specie of bug is rare.



Bugs: "What can we do to their Simper programs?"

"Debugging" is the process you use to find and change bugs. It's alot like playing detective. You must find out who did what, to whom, where and why. A good method of debugging is to pretend you are the Simper computer and obey each instruction. Start with the first one that Simper obeys when you RUN your program. Keep track of the values in the registers and memory locations and be sure you use the P register to tell you where the next instruction is in memory. Remember, the Simper computer adds 1 to the number in P just before it obeys the instruction it's looking at. Good luck!

program as saying: "It's true that you've typed 0", by stopping. You can also think of it as saying: "It's true that you have not typed 0", by not stopping. When you use JUMP, it's up to you to decide whether 0 in a register means true or false. You will find the JUMP operation to be very handy.

If the value in the B register isn't 0 (use JUMP to see), please type SCRATCH. You need this program in Simper's memory (use LIST to see):

```
001 :ASK A
002 :JUMP A 1
003 :HALT
```

If it isn't in memory, please type it in. Now we're going to put a "bug" in it. Sometimes a program of yours may not work the way you thought it would. Some people would say there is a bug in it. These bugs are neither the kind entomologists study, nor are they electronic spying devices. Bugs in programs cause unexpected things to happen. For example, your program might run on forever when you thought it should halt, or it might halt when you thought it shouldn't, or Simper might even complain: "ILLEGAL MEMORY REFERENCE". Here's the bug we're going to put into your program:

```
002 :JUMP B 1
```

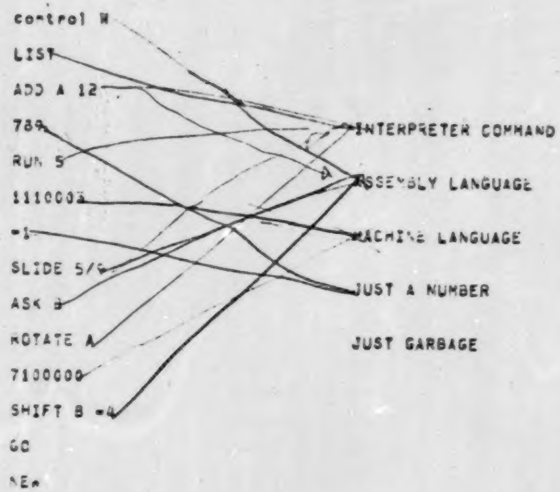
Ain't I purty ...... ?

Use FIX 2 to put the buggy instruction JUMP B 1 into location 2 in place of what's there now. Please LIST the program so you can inspect it.

Copyright 1974, A. B. Cannara and S. A. Weyer.

understand what ROTATE is doing, use control G to stop the program and use FIX to change the ROTATE instruction so that it moves digits to the right. Run that until you're satisfied, then replace the ROTATE with a SHIFT of your own choosing and RUN that until you're satisfied you understand the difference between SHIFT and ROTATE well.

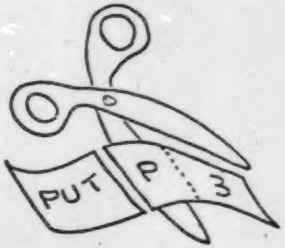
Now that you've had some experience in using Simper's commands, and assembly language and machine language instructions to write and run programs, please draw lines from the items in the lefthand column to their appropriate class on the right:


control W
LIST
ADD A 12
789
RUN 5
1110003
-1
SLIDE 5/6
ASK B
ROTATE A
7100000
SHIFT B -4
GO
NEW

INTERPRETER COMMAND
ASSEMBLY LANGUAGE
MACHINE LANGUAGE
JUST A NUMBER
JUST GARBAGE

Please show this to a tutor to make sure you understand how to classify the items in the lefthand column.

You probably remember that each Simper instruction, whether in assembly language (like PUT P 3) or in machine language (like 1120003), can have three parts (fields) count 'em!



Some instructions need only one field, the operation field, like HALT. Others need only the operation and register fields, like ASK. The rest need the address field as well. Some of these instructions use their address fields in special ways.

OPERATION REGISTER ADDRESS

Type HALT B 23 and an ENTER. What machine instruction (number) did Simper print to the right of your HALT instruction 7100000 ?
How many parts does it seem to have 1 ?

Does the Simper assembler seem to care if you type register and address fields in a HALT instruction?

NO

YES → LIST the HALT you just typed.

Does the assembler care about having address fieldsin ASK instructions?

NO

YES → Type ASK B 232 and an ENTER, and LIST it to see.

Remember that the Simper computer has a memory which has 250 locations. Any value from 1 to 250 can refer to one of those locations and when it does, it's called an address. Of the instructions which use an address field, four of them expect that field to contain something other than a normal address. They are PUT, EXCHANGE, SHIFT and ROTATE.

Copyright 1974, A. B. Cannara and S. A. Weyer.

Please SCRATCH to erase Simper's memory and type in this program:

```
001 :CASK B
002 :PUT P 1
```

It will read each character (letter, number, or punctuation) that you type on the typewriter. This program you've stored is a real life example of what mathematicians call a "function". A function is like a rule that lets you give an answer to something that someone tells you. For example, if you have a French to English dictionary, someone can give you a French word and you can give back the corresponding English word. The rule there is defined by the dictionary. It's an example of translation. The function (rule) that the program you just wrote uses is defined by the CASK operation. It translates each typewriter character (letters, digits and punctuation) into a particular 2-digit number and puts that number into the register you name in the instruction. Which register will that be in this program B...... ?

Please fill in the following table of characters and their numbers so that you can see what the CASK operation's function looks like. To do this, RUN your program with ENTER and observe the B register's values.

CASK translates this typewriter character (input)	into this number (output)
A	65.....
B	66.....
Z	90.....
9	57.....
8	56.....
.	46.....
@	64.....
/	47.....

Copyright 1974, A. B. Cannara
and S. A. Weyer.

Part 7

You have written (stored) programs in the Simper computer's memory just as if you had been writing a list of things down on paper for a friend to do for you when you told him to. You also have made Simper remember things (numbers) it doesn't understand as instructions. Remembering things and obeying a "stored program" are important. They make computers different from ordinary machines.

I remember what you want me to
SIMPER COMPUTER
002:
001:

The instructions that a computer can understand are important too. Not just any instructions will do. Fortunately, the instructions that Simper knows how to obey (see the list in Part 6) are more than enough to let you program the Simper computer to solve any problem that any other computer can solve.

Finally, a computer must always faithfully obey any program you write. It mustn't be fickle and decide not to obey your instruction in location 250 just because it's a long reach. It must have a clear way of knowing which instruction is to be obeyed next.

P: 130

I obey your your programs very carefully
SIMPER COMPUTER

Those three are basic abilities which make modern computers seem to satisfy Church's thesis (remember Part 17): that any problem a human can solve with pencil and paper can be solved by a computer. Or, in other words, if one person can tell another how to solve a problem, then he or she can tell a computer how to solve it.

Here's an old friend from the first test you took!

3	15
4	17
10	29

This table defines a function whose rule (program) you were asked to find. If you can't figure out the rule, ask a tutor. Now, please write a program to accept each value on the left as an input and output (type) the corresponding value on the right. Use SCRATCH to erase Simper's memory. Here are the instructions. All you need to do is put them into Simper's memory in the right order!

```
PUT 5 9         / GET THE NUMBER 9
STORE 6 100     / SAVE IT IN LOCATION 100
ASK A           / REQUEST AN INPUT NUMBER FROM HUMAN
STORE A 101     / SAVE IT IN 101
ADD A 101       / DOUBLE THE INPUT NUMBER
ADD A 100       / ADD 9 TO THE DOUBLED NUMBER
WRITE A         / TYPE OUT THE RESULT (OUTPUT) FOR HUMAN
PUT P 1         / DO IT ALL OVER AGAIN
```

Does this program produce the function table at the top of this page?

[✓ YES] NO

NO → Check your program or get help.

YES → Fill in some new inputs and outputs for the function your program has defined:

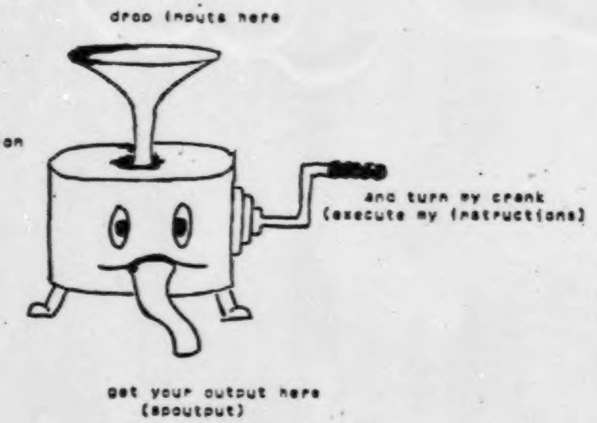
6	21
888	1795
47	103
49,302	98613

Type control G.

Could we change the PUT P 1 instruction to PUT P 3 and still get the same result?

[✓ YES] NO → Try it.

If you look at the table at the bottom of the second page in Part 6, you'll see that CASK can translate any typewriter key into a 2-digit number. Because of this, CASK is called a "total" function. When the Simper computer executes (obeys) your CASK instruction, the key you type is called the "input" to CASK. An input is what you give (tell) a function. All functions must have at least one input before they can decide what value to produce (output)!



Funky Function

You may remember that Simper accepts two kinds of literals. The first kind are numbers... (hint! see Part 3). The CASK operation makes the Simper computer accept typewriter characters as literals too. Why are they literals? Because CASK always produces the same number (output) for each character you type (input). So the table in Part 6 defines the CASK's function forever. If you memorized it, you would always know what number each key corresponds to. When Simper executes a CASK, the value for each typewriter key is no secret (remember Luigi Literal doesn't hide anything either).

Copyright 1974, A. B. Cannara and S. A. Weyer.

Please use the NAME command to make it easier to refer to the location that has 9 in it by typing: NAME 100 NINE and then FIX location 2 so that it is STORE B NINE. You can check that you have really named location 100 with the symbol NINE by typing another new command: NAMES and either ENTER or RETURN (try both to see what happens). What's the difference? [illegible] ....

Now use FIX to change location 6 to be ADD A NINE. LIST your program with ENTER.

Do the address fields of the two instructions you've changed still refer to the right memory location (the one that you want to have 9 in)?

[✓ YES] → Do you think it's easier to remember that the symbolic address NINE has the value 9 than to remember that location 100 (literal address) has the value 9? .yes.

NO → They should. Ask for help.

Now use NAME to give any name you please (with up to 6 letters) to location 101. Then use FIX to change the instructions in locations 4 and 5 so that they use that name (symbolic address). Use NAMES to see all the names you've defined. What name did you give to location 101? BOB............. What's that name's value right now? .R........

Names you make with NAME can be anything as long as they:

1. Start with a letter.
2. Contain only letters and numbers.
3. Have no more that six characters.

Why? Because 9 is already stored..

....

(ask for help if you can't answer).

Please change the PUT P 1 instruction to PUT P 3 if you haven't already.

After the instructions PUT B 9 and STORE B 100 have been executed once by the Simper computer, will the value in location 100 ever change?

[✓ NO] → What value is in location 100? ..9..... .

YES → LIST your program, RUN it again and ask for help.

Could the instructions PUT B 9 and STORE B 100 be changed to PUT A 9 and STORE A 100 without affecting the program?

[✓ YES] → Why? Because [illegible] ....

[illegible] in [illegible]

NO → Change them with FIX and RUN the program again.

What would happen if the PUT B 9 instruction was changed to PUT P 9?

It would [illegible]

....

....

(try it if you don't know, but before doing the next page, change the PUT P 9 back to what it was before, and check that the program works ok).

Put B 9
Store B 100
Fix A
[illegible]

Ask for
[illegible]
Put - 99
Run 99

Copyright 1974, A. B. Cannara and S. A. Weyer.

Part 9

Most computer languages that are like Simper have several types of addresses (names for memory locations). One kind may be more useful than another in a particular instruction or command. You've used literals (numbers) as addresses (like: ADD A 100 or LIST 5). You've also used symbols (like: ADD A NINE or DUMP NINE).

Let's see, everything is relative, so if I'm here then here + 3 is over there.
001
002
003
(HERE

Here's a new kind of address. It's called a "relative" address because it refers to a location (target) that is a certain distance away from a given location (it's related to the given location). The given location can be the location you are at now, for which you can use the special name ".", or any literal or symbolic address. The target is related to the given location by zero, or a positive or negative number. Here are some examples:

```
ADD A 100+3          SLIDE ./.+23
LIST NINE-4          DUMP 10/10+6
PUT P .+5            DIV B TWO-1
```

Please use SCRATCH to clear Simper's memory and then type the instruction PUT P . with an ENTER. What machine language instruction did the assembler produce? ..112001 Is the address part of it (right three digits) the same as the address of the location that PUT P . is in?

[✓ YES]
↓

NO
↓
It should be:

To Simper, a dot (.) in an address always means "the address I'm at now".

A new command: FORGET will help you get rid of names which your program no longer uses as addresses (references) in instructions. You can use FORGET to erase the name you used for location 101 before. First use NAMES to command the Simper interpreter to show you all the names you have created. Now type FORGET, a space, any name that your program isn't using as an address, then type a RETURN. Use NAMES again to check that what you forgot was really erased.

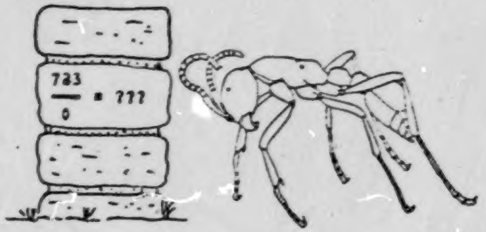
Now write a program on your own to define a different function from the one you have. You may want to SCRATCH your present program. Here are some sample formulas:

(input x input) - 3 = output

input1 + (input2 - 3) = output

2 x input1 / input2 = output

Notice that the last two formulas require that the program you write for them ASK two input values from you. The last formula requires that a program use DIVISION. In Simper, DIV really takes two memory locations, one for the DIV itself → DIV A NUMBER
and one immediately following location → PUT P YOOBIG
which the DIV makes the Simper computer obey
if you attempt to divide by zero (which in most computers is undefined). Usually a HALT in the location after a DIV is sufficient. Then your program will stop if there's a bug which might try a division by zero.



Copyright 1974, A. B. Cannara and S. A. Weyer.

Ok, now here's a few changes that you can make to your program which will make it destroy itself as it runs (a program with suicidal tendencies). Use [illegible], FIX and NAME to make your program look like:

```
001 $PUT A 10   / CHANGED
002 $PUT B 1    / NEW
003 $STO B ONE  / NEW
004 $ASK B
005 $STO B @A
006 $SUB A ONE  / NEW
007 $PUT P ,-3
```

Remember to NAME some location (like 100) ONE before you RUN this program. Use RUN and ENTER. Type any numbers you please to the ASK.

Did Simper finally stop your program with an "ERROR HALT"?

[✓ YES] → What message did Simper print when it tried to execute (obey) location 7? Unknown operation (C)........ LIST your program. Which instruction was destroyed by the last number you typed? Put P, -3........ What happened was that the Simper computer suddenly found the last number you typed in a location it was told by the P register to execute. It didn't know how to obey that number as an instruction. Please FIX location 7 so that it is PUT P ,-3 again. Now list your program with ENTER so you can see the machine language instruction in each location. Now RUN the program again with ENTER and instead of typing just any old numbers to the ASK, type the following sequence VERY carefully:

```
0
0
0
the machine language number in location 7
the machine language number in location 6
the machine language number in location 5
the machine language number in location 4
the machine language number in location 3
the machine language number in location 2
the machine language number in location 1
control G
```

NO → Type some more numbers or ask for help.

Do you think the program: 001 $PUT P , will run forever?

YES → Here is a program in which a relative address is handy. Please use SCRATCH and then type it in:

```
001 $PUT A 0          / MAKE A ZERO
002 $STORE A WORD     / USE IT TO EMPTY OUT WORD
003 $CASK A           / GET A CHARACTER FROM HUMAN
004 $LCR A WORD       / COMBINE IT WITH WHAT'S IN WORD
005 $ROT A L2         / MAKE ROOM FOR ANOTHER CHARACTER
006 $PUT P ,-4        / GO BACK FOR MORE
```

NO → Try it. RUN it with ENTER. ↑

Remember to NAME some location beyond 006 to be WORD. When you RUN this program, it will read in any 5-letter word you type letter-by-letter and carefully put all five letters into one memory location called WORD. Remember the character code table in Part 6. Each typewriter character is translated into a 2-digit number by CASK. Since a memory location can hold as many as ten digits, you can squeeze codes for up to five characters into any memory location (or register).

RUN the program with ENTER and carefully observe the activity in register A. Type any 5-letter word, one letter at a time. Notice how the ROTATE instruction makes room for each new character and how LCR combines each character with those you've typed before. After you've typed all five letters, stop the program with control G. ← What 5-letter word did you type? .... What is the value in location WORD? .... Please use the table in Part 6 to translate that value into five letters.

Are all the letters in WORD the same as those that you typed?

[✓ YES] ↓

NO → They should be! Don't worry about their order. ↑

Copyright 1974, A. B. Cannara and S. A. Weyer.

695

Here's a program that uses indirect addressing to make a code (letter substitution) for secret messages. Please clear Simper's memory with SCRATCH and type it in (you needn't type the remarks):

```
CASK B          / ASK HUMAN FOR A LETTER TO CODE
LOAD A #B       / USE ITS VALUE TO FIND ITS CODE LETTER
CWRITE A        / PRINT THE CODE LETTER
PUT P 1         / DO IT ALL AGAIN
```

Before you can use this you must put your substitution code into some memory locations. Where? Well, the first letter in the alphabet is A ......, so if you typed that to CASK B, register B ...... would have in it the numerical code for that letter, which is 65 .... Look at the table on the second page of Part 6. Now the LOAD instruction will use the value in the B ...... register as an adress ...., so in what location will your substitution code for the alphabet begin? 065 ....

Did you say "65"?

[✓ YES] ↓

NO ↓ Think about the program and the table in Part 6. Ask for help if you just don't understand.

Ok, FIX location 65 onward (type FIX 65/) and, using the table in Part 6, type into each location the numerical code of the character you wish to substitute for each letter in the alphabet. After you do this for the location whose address corresponds to "Z" (090), stop. Now you can RUN your program with RETURN or ENTER and type in letters to see what new letters your code substitutes for them. After you've typed some letters, type some other characters like "4" or "%".

Did your program print out a substitution character for any of the non-letters you typed?

[✓ NO] ↓

YES → Ask for help.

685

Now LIST your program with ENTER again. You should still have the same program in both assembly and machine language. Why? Because you simply typed all its machine language instructions (numbers) in backwards and it has stored them in just the right locations.

Did Simper stop your program with an error before you typed control G?

[✓ NO] ↓

YES ↓ Are you sure you typed the sequence exactly?

YES ↓ Ask for help.

NO ↓ FIX the program back up and try it again.

This should prove to you that the Simper computer only understands certain numbers as instructions. To really prove it, RUN the program again and make a slight change in one of the nonzero numbers in the sequence you typed.

Did you get Simper to complain "ERROR HALT" again?

[✓ YES] ↓

NO ↓ Try it again or ask for help.

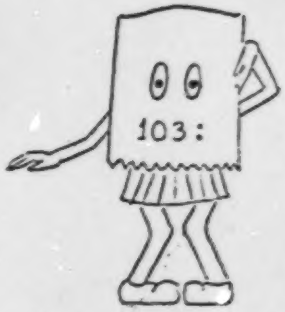
Good. Please LIST with ENTER again. You've proved a few things:

a. The Simper computer understands only certain numbers as instructions.

b. The Simper interpreter obeys LIST commands by translating machine language instructions (numbers) back into assembly language.

c. Indirect addressing is fun and helpful, but watch for bugs.

Copyright 1974, A. B. Cannara and S. A. Weyer.



You have been using addresses (literal, symbolic, relative or indirect) as names which have values. Often, these values have been numbers contained in memory locations, and you've been able to see them with FIX, DUMP or LIST, or use them in a program with LOAD, ADD, MULTIPLY, LDR etc. In the case of indirect addresses, you've used the name of a register (A, B or P) as a means for finding a value which your program can use as an address. The interesting thing about indirect addressing is that it can be extended without limit. So names (location addresses) can have values (numbers in them) which can themselves be used as addresses (names) of locations which in turn have values which can be used as addresses which have values which ... and on and on and on ... . This is as if you wanted to visit your friend Lilac whose home address you don't know, but you do know that your friend Gerard knows her address, but you've forgotten his address, so you go over to Paula's house to ask her where he lives (gasp). Here's a program. Use SCRATCH and then type it in. Remember to name location 57 PAULA and FIX the values of locations 20, 57 and 103 to be as shown:

```
001 :PUT B PAULA          / GET PAULA'S ADDRESS
002 :LOAD B @B            / GO GET GERARD'S ADDRESS (PAULA'S VALUE)
003 :LOAD B @B            / GET LILAC'S ADDRESS (GERARD'S VALUE)
004 :LOAD B @B            / GET HER VALUE (HER MESSAGE TO YOU)
005 :CWRITE B             / PRINT
006 :SHIFT B R2           / IT
007 :CWRITE B             / OUT
008 :HALT

020 :103

057 :20 (PAULA)

103 :7372
```

Before you RUN this program, what do you think Lilac's message is? Hi ......

What's her address? ...... What's Gerard's address? ...... What's your address? ...... RUN the program with ENTER and again with RETURN. What was Lilac's message? ......

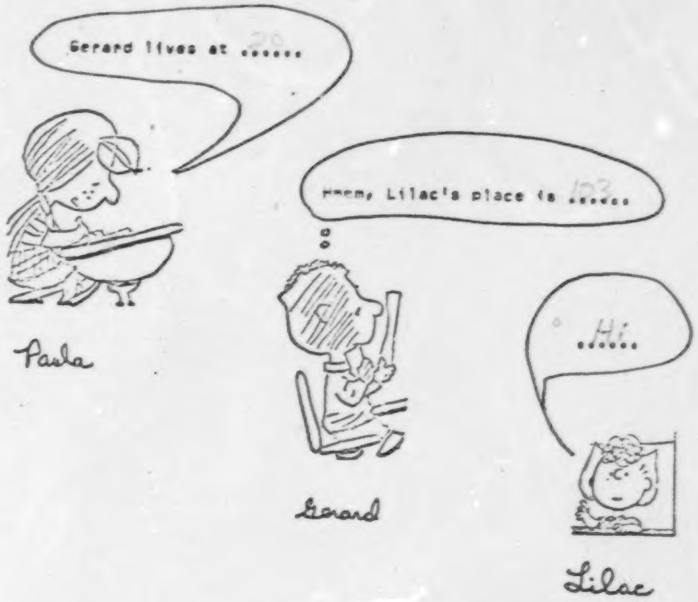
Was it what you expected?

YES

NO

Think about what each instruction in the program does and RUN it again.

You are now a licensed indirect addresser.



Copyright 1974, A. B. Cannara and S. A. Weyer.

You may have noticed that the procedures you used in your code program don't seem to want any return address. That's because they never expect to return to the place from which they were called. They were designed run forever or until you stop them with control G. Furthermore, neither

```
CAS B          nor    PUT A START
LOA A #B              CHR A
CHR A                 LOA B #A
PUT P 1               CHR B
                      INC A
                      PUT B 32
                      CHR B
                      PUT P .-6
```

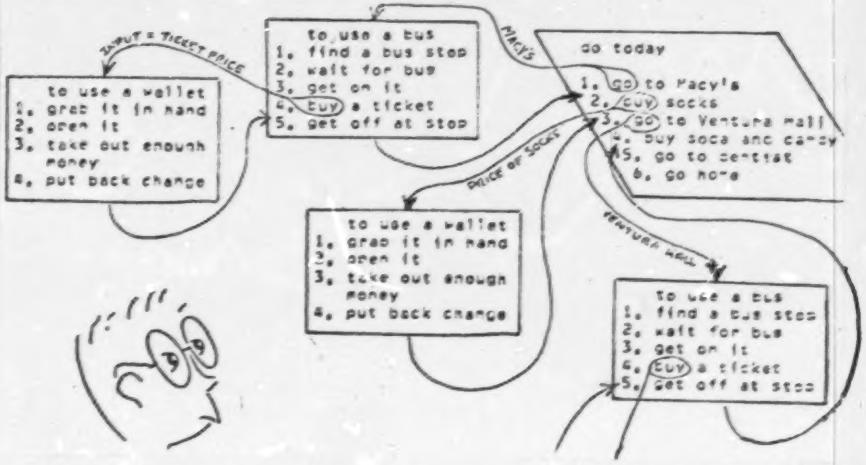
expect inputs when they are called. However, the sub-procedure expects an input in the A register which it will use as a character code and an address. These are degenerate procedures because they never use a return address.

Whenever you write a procedure, you determine the calling sequence which tells that procedure its return address and inputs. Each procedure can have a different calling sequence. Whenever you call a procedure, you must write instructions which produce the calling sequence that particular procedure expects. Then you "transfer control" to the procedure, which just means that you transfer the computer's attention to the instructions in that procedure. You already know how to do this: simply PUT the address of the procedure's first instruction into the P register and the Singer computer will immediately begin executing its instructions.

Let's write a procedure with a calling sequence which expects a return address and two inputs, and returns its result (output) in the A register. The procedure should SHIFT the second input left two digits and then LOR the right two digits of the first input into that, making its output value. You will have use for this procedure as part of a poster-writing program.

Let's see if you understand how the bus and wallet procedures were used when you obeyed your list of things to do. When you did line 1 of your list, you called the bus procedure, gave it the input (destination) ...... and told it to return when it was done so you could finish line .... At its line 4, the bus procedure called the wallet .... procedure, gave it an input which was the price of the ticket and told it to return to finish line 4 ...... of the ...... procedure. When the bus procedure finished, you were at ...... ready to buy socks ...... So you called the wallet .... procedure in order to pay for them. With that done, you called the ...... procedure again so you could go to dentist .... And so on, until you reached home ....

You may have noticed one good reason for using procedures. They save time and space if they are used very often because you only need to write them once in your program. Whenever you call them you tell them where to come back. It's as if the whole procedure were automatically inserted at each place you used (called) it:



Copyright 1974, A. B. Cannara and S. A. Weyer.

Please use SCRATCH to erase memory. NAME some location like 50 or 100 to be SHIFTY. That will be the procedure's name. Now FIX SHIFTY/ and we'll define the calling sequence for this procedure. It will expect two inputs and a return address, so we need three open locations in the procedure for storing these pieces of information (the "calling parameters"). They won't be instructions. Why should the procedure make Simper skip over them? ....

```
PUT P .+4 (SHIFTY)       / SKIP OVER CALLING PARAMETERS
0                        / PLACE FOR RETURN ADDRESS
0                        / PLACE FOR INPUT #1
0                        / PLACE FOR INPUT #2
```

For this procedure, let's agree that the first location after its first instruction will contain the return address, the second will have the first input, and the third will have the second input. Now type the PUT P .+4 instruction and the three zeros into SHIFTY's first four locations. In the next few locations, write these instructions which actually do the work of the procedure:

```
LOA A SHIFTY+3           / GET INPUT #2
SHI A L2                 / SHIFT IT LEFT 2 DIGITS
LOR A SHIFTY+2           / COMBINE IT WITH INPUT #1
LOA P SHIFTY+1           / RETURN WITH RESULT IN A
```

Now that you've defined your first genuine procedure, how do you use (call) it? Well, you know it expects that three things (calling parameters) be stored in it before it can do its work. So, wherever you call it in your program, you must at least have three STORE instructions which give it the inputs and the return address. You will need two more instructions, one which makes the return address and one which transfers control to SHIFTY. Here's a typical way to call SHIFTY, assuming register A has input #1 in it and B has input #2:

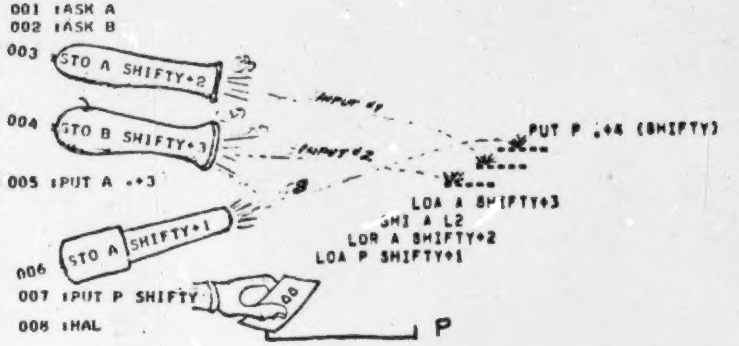
```
STO A SHIFTY+2   / TELL SHIFTY INPUT #1
STO B SHIFTY+3   / TELL SHIFTY INPUT #2
PUT A .+3        / MAKE THE RETURN ADDRESS
STO A SHIFTY+1   / TELL SHIFTY ABOUT IT
PUT P SHIFTY     / TRANSFER CONTROL TO SHIFTY
```

wherever you want to call (engage the services of) SHIFTY in your program, you must use five instructions like those above. Now let's write a program that will get two inputs from you and then call SHIFTY. Type FIX 1/ and these two instructions:

```
001 :ASK A       / GET INPUT #1 FROM HUMAN
002 :ASK B       / GET INPUT #2 FROM HUMAN
```

Now type in the five instructions above which produce the calling sequence. Then type a HALT. LIST your program so you can see everything. Now RUN your program with ENTER. Type any 2-digit number for the first "INPUT NUMBER:" and any number but zero for the second. Observe what happens when SHIFTY is called, especially the P register's value. When your program stops, LIST it again. What value is in location SHIFTY+1 ? 4 ......, in SHIFTY+2 ? 12 ......, in SHIFTY+3 ? [illegible] ......
Are these values the correct return address and inputs?

YES ↓ NO → Ask for help.



```
001 :ASK A
002 :ASK B
003 STO A SHIFTY+2
004 STO B SHIFTY+3
005 :PUT A .+3
006 STO A SHIFTY+1
007 :PUT P SHIFTY
008 :HAL
```

```
PUT P .+4 (SHIFTY)
*---
*---
*---
LOA A SHIFTY+3
SHI A L2
LOR A SHIFTY+2
LOA P SHIFTY+1
```

Copyright 1974, A. B. Cannara and S. A. Weyer.

In this program, the decision to stop was made by the JUMP instruction. The stopping rule was simple: if the B register's value was zero, it was time to stop typing characters. In other words, the answer to the question: "Is it time to stop?", was "True" if there was a zero in the B register when JUMP was executed. Otherwise the answer was "False". You may remember, when you first saw JUMP in Part 5, that we said it is up to you to decide whether zero in a register means "true" or "false" when you use JUMP to make a decision.

TRUE AGAIN!
FALSE!

```
               CAS A
The program:   ASK B    you've been using has a bug.
               NEG B
               JUMP B ,+2 (AGAIN)
               HAL
               CHR A
               INC B
               PUT P AGAIN
```

Please RUN it again with ENTER, type a character and then a small negative number, like -2 or -3. Notice what happens to the B register's value as the program runs.

Do you think the program will ever stop?

NO

YES

Check that you have the program above.

The number you typed was ...... and the NEGATE instruction changed it to ...... Each time the character was printed, INCREMENT added ...... to the B register's value. The stopping rule that worked before will not work for negative input numbers because the B register's value will never become ......, so the JUMP instruction will always cause Simper to go ahead and execute the .... instruction.

Part 12

You are now experienced in writing procedures that can be called from any place in your programs. And you are almost ready to write programs which make the typewriter print signs, pictures and anything else you can think up.

The next thing to learn is how to make a program (procedure) stop itself as soon as it finishes its job. You must write instructions into your programs which decide when the programs should stop and when they should keep working. You know one operation which can be used for making decisions. It is JUMP.

JUMP decides whether the Simper computer will execute the next instruction in a program, or an instruction at any other place in the program. If you forget how JUMP works look at the table in Part 6.

Simper has another operation to help your programs make decisions. It is COMPARE. It looks at two numbers, one that is in a register and one that is in a memory location. The instruction that Simper executes next depends on the values of the two numbers that COMPARE looked at.

The process of doing something over and over until it has been done just the right number of times is called "iteration" or recursion with a "stopping rule". Most of the programs you've written either executed all their instructions just once and stopped, or executed them forever (until you got tired and typed control G).

Now let's write two short programs. Both will do the same

Copyright 1974, A. B. Cannara and S. A. Weyer.

The bug in the first program is easily cured by using COMPARE instead of JUMP. This is the second program. Please use FIX and SLIDE to change the previous program so it is:

```
001 :CASK A              / ASK HUMAN WHAT CHARACTER IS TO BE TYPED
002 :ASK B               / ASK HUMAN HOW MANY TIMES TO TYPE IT
003 :STORE B COUNT       / SAVE THAT IN A LOCATION CALLED COUNT
004 :PUT B 0             / CLEAR THE B REGISTER
005 :COMPARE B COUNT     / TYPE SOME MORE?
006 :NOP                 / NO!
007 :HALT                / NO!
008 :CWRITE A            / YES!
009 :INCREMENT B         / CHARACTER HAS BEEN TYPED ONCE MORE
010 :PUT P AGAIN         / GO SEE IF ENOUGH HAVE BEEN TYPED
```

Before running this program, be sure to NAME location 5 to be AGAIN and NAME some location outside your program to be COUNT. Ok, RUN it and try to find out how wide the typewriter paper is. What's the largest number of characters your program can type on one line? 72 ...... RUN it several times and try typing a negative number to the ASK.

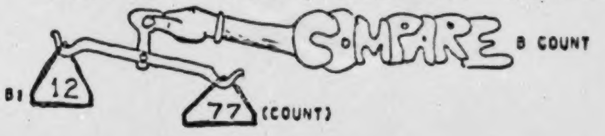
Did the program stop even for a negative number?

[✓ YES] NO

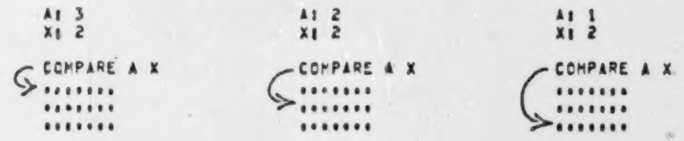
Check that your program is the one above.

That's because COMPARE is more powerful than JUMP. COMPARE does what its name suggests. It looks at two numbers, one in a register, the other in a memory location and compares their values. It decides which one, of three possible statements about the two numbers, is true:

1. The number in memory is smaller than the number in the register.
2. The number in memory is equal to the number in the register.
3. The number in memory is greater than the number in the register.



If statement #1 is true, COMPARE does nothing. It allows the Simper computer to execute the instruction which immediately follows the COMPARE. If statement #2 is true, COMPARE makes Simper skip the next instruction. If statement #3 is true, COMPARE makes Simper skip the next two instructions. For example, if register A and location X have the values shown below, then a COMPARE instruction will cause Simper to skip or not skip, as shown here:



The program on the previous page used this sequence of instructions:

```
COMPARE B COUNT
NOP
HALT
CWRITE
```

The stopping rule which this produced was "stop when the value in B is greater than or equal to the value in COUNT". The "greater than" part of the rule is what killed the bug that bit the first program. In the second program, if you typed a negative number to the ASK, it went into register ...... and was stored in location ...... The PUT instruction made B have the value ...... The COMPARE B COUNT then saw that the value in COUNT was .... than the value in B. That's possibility # ...... on the previous page, so the next instruction Simper executed was NOP. But NOP stands for "no operation" or "do nothing", so the next instruction ...... was executed and that stopped the program.

When you ran the second program and you typed a positive number, it was also stored in ...... The value in the ...... register also started at ...... But now the COMPARE instruction saw that the value

Copyright 1974, A. B. Cannara and S. A. Weyer.

Part 13

Now let's write a poster program that can enlarge any letter in the alphabet in any shape you wish. To do this, you must understand how to define a procedure and its calling sequence (see Part 11), how to use relative and indirect addresses (see Parts 9 and 10), and how Simper executes JUMP and COMPARE (see Part 12).

Please use SCRATCH to erase any program you might now have in memory. We'll build your program piece-by-piece. Here's the first part; it will type out exactly five characters from a memory location using ROTATE (see Part 9):

```
PUT A 5          / MAKE A 5 FOR COUNTING CHARACTERS
STO A FIVE       / SAVE IT
LOA A STRING     / GET THE CHARACTERS TO TYPE
PUT B 0          / CLEAR THE B REGISTER
COM B FIVE       / TYPED 5 CHARACTERS YET?
NOP              / YES!
HALT             / YES!
ROT A L2         / GET NEXT CHARACTER READY
CHR A            / TYPE IT
INC B            / KEEP COUNT
PUT P MCRE       / GO BACK FOR MORE CHARACTERS
```

Before you run this program, NAME the location containing the COMPARE to be MORE, then NAME two locations outside your program FIVE and STRING. Finally, FIX STRING and type a number there that contains the codes (see Part 6) for any five characters. Now RUN your program with ENTER and observe the values in the registers.

Did your program stop itself after typing all five characters in STRING?

YES ↓

NO ↓ Check your program or ask for help.

What's the stopping rule that it uses? ....

JUMP. It is still up to you to determine whether any decision made by a COMPARE instruction means "true" or "false" in your programs.

JUMP and COMPARE allow your programs to make decisions and so change their "minds" about what to do next. This ability is called "program control". It is one of the most important abilities that any general-purpose computer must have. A computer must be able to make decisions, store and execute a program in its memory, and know enough instructions before it can be used to play games, draw pictures or answer questions.

The two previous programs made decisions about stopping what they were doing. In this case, since they were only doing one thing (typing the character you asked for), stopping that meant stopping the whole program. That doesn't always have to be the case. Some other programs you'll write will do one thing for a while, then decide to stop that by using JUMP or COMPARE, and then do something else until a decision is made to stop that, etc.

Remember that the process of doing something over and over is called iteration or recursion. When what is being done has a goal, then there is some stopping rule which decides when the goal is reached. The two programs you've used in this section had the same goal. It was to type out the character you wanted as many times as you wanted. Both programs used iteration to reduce to zero the difference between the goal (the number you typed) and the number of characters printed. The goal

Copyright 1974, A. B. Cannara and S. A. Weyer.

222

Messages stored this way are called "strings" because their characters occur in memory one after the other, like beads on a string. Strings have beginnings (their first character) and ends (their last). Even though they are stored in Simper's memory as numbers, your TYPE procedure translates them into characters. So what TYPE does is give Simper a new way of storing information. Information is often called "data", so now you can make Simper programs that "understand" two kinds of data: strings and numbers. A method for storing data is called a "data-structure". Simper normally lets you use the data-structure of ten-digit numbers. TYPE lets you use string data-structures too.

Is TYPE a function?

YES ↘ NO ↓

Doesn't it accept an input (a string) and produce an output (print the string on the typewriter)?

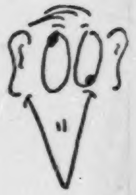
One very important thing about programming computers is that functions can define new data-structures. Since you define functions by writing procedures, you can make any computer use any data-structure you please. Please look up "data" and "structure" in a dictionary.

Save your program if you wish. Now write a procedure that is alot like TYPE except that, instead of typing out characters, it reads them in from you and stores them away, five per location, in a sequence of locations. You may also make it so you can type a special character (like ENTER = 27) to tell it to end the string with a zero. Then if you combine this with your TYPE procedure, you can load up messages and posters by typing them directly, rather than by typing ten-digit numbers into memory locations.

Your TYPE procedure is almost ready to be used. Notice that it has two iterative parts. Each part has a different goal and so each has a different stopping rule.

When TYPE is called (executed), its first part (George) finds the location he was given as an input and gets from that his first load of characters. George gives them, as a number in the A register, to TYPE's second part (Martha) who types out exactly five characters from that register. When Martha finishes with one register full of characters, she tells George to get another load (PUT P NEXT). George has already made the address of the next location where he will look for characters: he added 1 to the value in TYPE+2 with the INC B and STO B TYPE+2 instructions. He's ready to see if he can get another location with five characters in it. If he gets an empty location (0), he'll stop and use the return address to go back where he was called.

```
100 :PUT P .+3 (TYPE)

103 :LOA B TYPE+2 (NEXT)
104 :LOA A @B
105 :JUM A .+2
106 :LOA P TYPE+1
107 :INC B
108 :STO B TYPE+2
                    George
```



```
109 :PUT B 0
110 :COM B FIVE (MORE)
111 :NOP
112 :PUT P NEXT
113 :ROT A L2
114 :CWR A
115 :INC B
116 :PUT P MORE
                    Martha
```

Martha is good at counting up to any number because she uses a COMPARE instruction. George doesn't care how far he has to count. He just keeps on looking at one location after another until he finds an empty one (JUMP A .+2). They both iterate to reach their goals.

What's George's stopping rule? ....

What's Martha's? ....

Copyright 1974, A. B. Cannara and S. A. Weyer.

Acknowledgements

This work was supported by National Science Foundation Grant NSF-GJ-443X. We are further indebted to our student volunteers. They remain anonymous, but they were the most important people in the experiment. We also thank Adele Goldberg and Diane Kanerva for their editorial assistance.

References

Atkinson, R., Fletcher, J. D., Lindsay, E., Campbell, O., & Barr, A. *Computer-assisted instruction in initial reading*. (Tech. Rep. No. 207) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Baecker, R. Personal communication, 1974.

Benbasset, G., & Sanders, W. Personal communication, 1974.

Berry, P. Pretending to have (or to be) a computer as a strategy in teaching. *Harvard Educational Review*, 1964, *34*, 383-401.

Bradley, J. *Distribution-free statistical tests*. Englewood Cliffs, N.J.: Prentice-Hall, 1968.

Brand, S. *Two cybernetic frontiers*. New York/Berkeley Calif.: Random House/The Bookworks, 1974.

Bredt, T. *A computer model of information processing in children*. (Tech. Rep. No. CS100) Stanford, Calif: Computer Science Department, Stanford University, 1968.

Brown, J., & Burton, R. SOPHIE -- A pragmatic use of AI in CAI. The ACM National Conference, San Diego, 1974.

Brown, J., & Rubinstein, R. *Recursive functional programming for students in the humanities and social sciences*. (Report No. 27) Irvine, Calif.: Department of Information and Computer Science, U. C. Irvine, 1973.

Cannara, A. *Children learning computer programming*. Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1975, forthcoming.

Cannara, A., & Weyer, S. A study of children's programming. The 1974 Conference on Computer-Based Learning Systems, University of Hamburg, Federal Republic of Germany, 1974.

Carbonell, J. *Mixed initiative man-computer instruction*. (Report No. 1971) Boston: Bolt, Beranek & Newman, 1970.

Davis, M. (Ed.) *The undecidable*. New York: Raven Press, 1965.

Denning, P. Virtual memory. *Computing Surveys*, 1970, *2*, 153-189.

Dwyer, T. A. *An experiment in the regional use of computers by secondary schools*. Final Report, NSF-OCA-GJ1077-SOLO, 1972.

Ellis, A. The use and misuse of computers in education. New York: McGraw-Hill, 1972.

Evey, R. J. The theory and applications of pushdown store machines. Unpublished doctoral dissertation, Harvard University, 1963.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. Programming languages as a conceptual framework for teaching mathematics. (Report No. 1889) Boston: Bolt, Beranek & Newman, 1969.

Feurzeig, W., Lukas, G., Faflick, P., Grant, R., Lukas, J., Morgan, C., Weiner, W., & Wexelblat, P. Programming languages as a conceptual framework for teaching mathematics. (Report No. 2165) Final Report, NSF-C-615, Vols. 1-3. Boston: Bolt, Beranek & Newman, 1971.

Feurzeig, W., & Lukas, G. Logo: A programming language for teaching mathematics. Educational Technology, March, 1972.

Feurzeig, W., & Lukas, G. A programmable robot for teaching. The International Congress of Cybernetics and Systems, Oxford, England, 1972.

Fischer, G. Material and ideas to teach an introductory programming course using Logo. Irvine, Calif.: Department of Information and Computer Science, U. C. Irvine, 1973.

Folk, M., Statz, J., & Seidman, R. Syracuse university Logo project (Report No. 3) Final Report, NSF-TIE-GJ32222-3. Syracuse, New York: Syracuse University, 1974.

Goldberg, A. Computer-assisted instruction: The application of theorem proving to adaptive response analysis. (Tech. Rep. No. 203). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Goldberg, A., Levine, D., & Weyer, S. Three sample instructional programs from Stanford University. Computers in the instructional process: Report of an international school. Ann Arbor, Mich.: Extend Publications, 1974.

Kay, A. A personal computer for children of all ages. The ACM National Conference, Boston, 1972.

Kay, A. A dynamic medium for creative thought. Meeting of The National Council of Teachers of English, Minneapolis, 1972.

Kimball, R. Self-optimizing computer-assisted tutoring: theory and practice. (Tech. Rep. No. 206). Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Knuth, D. *MIX*. Reading, Mass.: Addison-Wesley Series in Computer Science and Information Processing, 1970.

Koestler, A. *The roots of coincidence*. New York: Vintage Books/Random House, 1973.

Levison, M., Ward, G., & Webb, J. *The settlement of Polynesia. A computer simulation*. Minneapolis: University of Minnesota Press, 1973.

Lorton, P., & Slimick, J. Computer-based instruction in computer programming. Fall Joint Computer Conference, Las Vegas, 1969.

Manis, V. *A machine independent implementation of Logo*. Unpublished doctoral dissertation, University of British Columbia, 1973.

Manna, Z. *Introduction to the mathematical theory of computation*. New York: McGraw-Hill, 1972.

Milner, S. The effects of computer programming on performance in mathematics. Annual Meeting of the AERA, New Orleans, February, 1973.

Minsky, M. *Computation: Finite and infinite machines*. Englewood Cliffs, N.J.: Prentice-Hall, 1967.

Oettinger, A., & Marks, S. *Run, computer run: The mythology of educational innovation*. Boston: Harvard Press, 1969.

Papert, S. Teaching children thinking. IFIP Conference on Computer Education, Amsterdam, August, 1970.

Piaget, J. *Genetic epistemology*. New York: Columbia University Press, 1970.

Polya, G. *How to solve it*. Princeton, N.J.: Princeton University Press, 1957.

Puri, M. (Ed.) *Nonparametric techniques in statistical inference*. London: Cambridge University Press, 1970.

Roman, R. *Logo: A student manual*. Pittsburgh: Learning Research and Development Center, University of Pittsburgh, 1972.

Scribner, S., & Cole, M. Cognitive consequences of formal and informal education. *Science*, *182*(4112), 9 November, 1973.

Smallwood, R. *A decision structure for teaching machines*. Boston: MIT Press, 1962.

Swinehart, D., & Sproull, R. *SAIL*. (Sailon No. 57.2), Stanford, Calif.: Stanford Artificial Intelligence Laboratory, 1971.

Toomre, A., & Toomre, J. Violent tides between galaxies. Scientific American, 229(6), December, 1973.

Winograd, T. Procedures as a representation of data in a computer program for understanding natural language. (Project MAC TR-84), Boston: MIT, 1971.

Winograd, T. When will computers understand people? Psychology Today, May, 1974.

Wittrock, M. (Ed.) Changing education: Alternatives from educational research. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Worthen, B., & Sanders, J. Educational evaluation: Theory and practice. Worthington, Ohio: Charles Jones Publishing Co., 1973.

(Continued from inside front cover)

165 L. J. Hubert. A formal model for the perceptual processing of geometric configurations. February 19, 1971. (A statistical method for investigating the perceptual confusions among geometric configurations. Journal of Mathematical Psychology, 1972, 9, 389-403.)

166 J. F. Juola, I. S. Fischler, C. T. Wood, and R. C. Atkinson. Recognition time for information stored in long-term memory. (Perception and Psychophysics, 1971, 10, 8-14.)

167 R. L. Klatzky and R. C. Atkinson. Specialization of the cerebral hemispheres in scanning for information in short-term memory. (Perception and Psychophysics, 1971, 10, 335-338.)

168 J. D. Fletcher and R. C. Atkinson. An evaluation of the Stanford CAI program in initial reading (grades K through 3). March 12, 1971. (Evaluation of the Stanford CAI program in initial reading. Journal of Educational Psychology, 1972, 63, 597-602.)

169 J. F. Juola and R. C. Atkinson. Memory scanning for words versus categories. (Journal of Verbal Learning and Verbal Behavior, 1971, 10, 522-527.)

170 I. S. Fischler and J. F. Juola. Effects of repeated tests on recognition time for information in long-term memory. (Journal of Experimental Psychology, 1971, 91, 54-58.)

171 P. Suppes. Semantics of context-free fragments of natural languages. March 30, 1971. (In K. J. J. Hintikka, J. M. E. Moravcsik, and P. Suppes (Eds.), Approaches to natural language. Dordrecht: Reidel, 1973. Pp. 221-242.)

172 J. Friend. INSTRUCT coders' manual. May 1, 1971.

173 R. C. Atkinson and R. M. Shiffrin. The control processes of short-term memory. April 19, 1971. (The control of short-term memory. Scientific American, 1971, 224, 82-90.)

174 P. Suppes. Computer-assisted instruction at Stanford. May 19, 1971. (In Man and computer. Proceedings of international conference, Bordeaux, 1970. Basel: Karger, 1972. Pp. 298-330.)

175 D. Jamison, J. D. Fletcher, P. Suppes, and R. C. Atkinson. Cost and performance of computer-assisted instruction for education of disadvantaged children. July, 1971.

176 J. Offir. Some mathematical models of individual differences in learning and performance. June 28, 1971. (Stochastic learning models with distribution of parameters. Journal of Mathematical Psychology, 1972, 9(4),)

177 R. C. Atkinson and J. F. Juola. Factors influencing speed and accuracy of word recognition. August 12, 1971. (In S. Kornblum (Ed.), Attention and performance IV. New York: Academic Press, 1973.)

178 P. Suppes, A. Goldberg, G. Kanz, B. Searle, and C. Stauffer. Teacher's handbook for CAI courses. September 1, 1971.

179 A. Goldberg. A generalized instructional system for elementary mathematical logic. October 11, 1971.

180 M. Jerman. Instruction in problem solving and an analysis of structural variables that contribute to problem-solving difficulty. November 12, 1971. (Individualized instruction in problem solving in elementary mathematics. Journal for Research in Mathematics Education, 1973, 4, 6-19.)

181 P. Suppes. On the grammar and model-theoretic semantics of children's noun phrases. November 29, 1971.

182 G. Kreisel. Five notes on the application of proof theory to computer science. December 10, 1971.

183 J. M. Moloney. An investigation of college student performance on a logic curriculum in a computer-assisted instruction setting. January 28, 1972.

184 J. E. Friend, J. D. Fletcher, and R. C. Atkinson. Student performance in computer-assisted instruction in programming. May 10, 1972.

185 R. L. Smith, Jr. The syntax and semantics of ERICA. June 14, 1972.

186 A. Goldberg and P. Suppes. A computer-assisted instruction program for exercises on finding axioms. June 23, 1972. (Educational Studies in Mathematics, 1972, 4, 429-449.)

187 R. C. Atkinson. Ingredients for a theory of instruction. June 26, 1972. (American Psychologist, 1972, 27, 921-931.)

188 J. D. Bonvillian and V. R. Charrow. Psycholinguistic implications of deafness: A review. July 14, 1972.

189 P. Arabie and S. A. Boorman. Multidimensional scaling of measures of distance between partitions. July 26, 1972. (Journal of Mathematical Psychology, 1973, 10,)

190 J. Ball and D. Jamison. Computer-assisted instruction for dispersed populations: System cost models. September 15, 1972. (Instructional Science, 1973, 1, 469-501.)

191 W. R. Sanders and J. R. Ball. Logic documentation standard for the Institute for Mathematical Studies in the Social Sciences. October 4, 1972.

192 M. T. Kane. Variability in the proof behavior of college students in a CAI course in logic as a function of problem characteristics. October 6, 1972.

193 P. Suppes. Facts and fantasies of education. October 18, 1972. (In M. C. Wittrock (Ed.), Changing education: Alternatives from educational research. Englewood Cliffs, N. J.: Prentice-Hall, 1973. Pp. 6-45.)

194 R. C. Atkinson and J. F. Juola. Search and decision processes in recognition memory. October 27, 1972.

195 P. Suppes, R. Smith, and M. Léveillé. The French syntax and semantics of PHILIPPE, part 1: Noun phrases. November 3, 1972.

196 D. Jamison, P. Suppes, and S. Wells. The effectiveness of alternative instructional methods: A survey. November, 1972.

197 P. Suppes. A survey of cognition in handicapped children. December 29, 1972.

198 B. Searle, P. Lorton, Jr., A. Goldberg, P. Suppes, N. Ledet, and C. Jones. Computer-assisted instruction program: Tennessee State University. February 14, 1973.

199 D. R. Levine. Computer-based analytic grading for German grammar instruction. March 16, 1973.

200 P. Suppes, J. D. Fletcher, M. Zanotti, P. V. Lorton, Jr., and B. W. Searle. Evaluation of computer-assisted instruction in elementary mathematics for hearing-impaired students. March 17, 1973.

201 G. A. Huff. Geometry and formal linguistics. April 27, 1973.

202 C. Jensema. Useful techniques for applying latent trait mental-test theory. May 9, 1973.

203 A. Goldberg. Computer-assisted instruction: The application of theorem-proving to adaptive response analysis. May 25, 1973.

204 R. C. Atkinson, D. J. Herrmann, and K. T. Wescourt. Search processes in recognition memory. June 8, 1973.

205 J. Van Campen. A computer-based introduction to the morphology of Old Church Slavonic. June 18, 1973.

206 R. B. Kimball. Self-optimizing computer-assisted tutoring: Theory and practice. June 25, 1973.

207 R. C. Atkinson, J. D. Fletcher, E. J. Lindsay, J. O. Campbell, and A. Barr. Computer-assisted instruction in initial reading. July 9, 1973.

208 V. R. Charrow and J. D. Fletcher. English as the second language of deaf students. July 20, 1973.

209 J. A. Paulson. An evaluation of instructional strategies in a simple learning situation. July 30, 1973.

210 N. Martin. Convergence properties of a class of probabilistic adaptive schemes called sequential reproductive plans. July 31, 1973.

(Continued from inside back cover)

211 J. Friend. Computer-assisted instruction in programming: A curriculum description. July 31, 1973.

212 S. A. Weyer. Fingerspelling by computer. August 17, 1973.

213 B. W. Searle, P. Lorton, Jr., and P. Suppes. Structural variables affecting CAI performance on arithmetic word problems of disadvantaged and deaf students. September 4, 1973.

214 P. Suppes, J. D. Fletcher, and M. Zanotti. Models of individual trajectories in computer-assisted instruction for deaf students. October 31, 1973.

215 J. D. Fletcher and M. H. Beard. Computer-assisted instruction in language arts for hearing-impaired students. October 31, 1973.

216 J. D. Fletcher. Transfer from alternative presentations of spelling patterns in initial reading. September 28, 1973.

217 P. Suppes, J. D. Fletcher, and M. Zanotti. Performance models of American Indian students on computer-assisted instruction in elementary mathematics. October 31, 1973.

218 J. Fiksel. A network-of-automata model for question-answering in semantic memory. October 31, 1973.

219 P. Suppes. The concept of obligation in the context of decision theory. (In J. Leach, R. Butts, and G. Pearce (Eds.), *Science, decision and value*. (Proceedings of the fifth University of Western Ontario philosophy colloquium, 1969.) Dordrecht: Reidel, 1973. Pp. 1-14.)

220 F. L. Rawson. Set-theoretical semantics for elementary mathematical language. November 7, 1973.

221 R. Schupbach. Toward a computer-based course in the history of the Russian literary language. December 31, 1973.

222 M. Beard, P. Lorton, B. W. Searle, and R. C. Atkinson. Comparison of student performance and attitude under three lesson-selection strategies in computer-assisted instruction. December 31, 1973.

223 D. G. Danforth, D. R. Rogosa, and P. Suppes. Learning models for real-time speech recognition. January 15, 1974.

224 M.R. Raugh and R. C. Atkinson. A mnemonic method for the acquisition of a second-language vocabulary. March 15, 1974.

225 P. Suppes. The semantics of children's language. (*American Psychologist*, 1974, *29*, 103-114.)

226 P. Suppes and E. M. Gammon. Grammar and semantics of some six-year-old black children's noun phrases.

227 N. W. Smith. A question-answering system for elementary mathematics. April 19, 1974.

228 A. Barr, M. Beard, and R. C. Atkinson. A rationale and description of the BASIC instructional program. April 22, 1974.

229 P. Suppes. Congruence of meaning. (*Proceedings and Addresses of the American Philosophical Association*, 1973, *46*, 21-38.)

230 P. Suppes. New foundations of objective probability: Axioms for propensities. (In P. Suppes, L. Henkin, Gr. C. Moisil, and A. Joja (Eds.), *Logic, methodology, and philosophy of science IV: Proceedings of the fourth international congress for logic, methodology and philosophy of science, Bucharest*, 1971. Amsterdam: North-Holland, 1973. Pp.515-529.)

231 P. Suppes. The structure of theories and the analysis of data. (In F. Suppe (Ed.), *The structure of scientific theories*. Urbana, Ill.: University of Illinois Press, 1974. Pp. 267-283.)

232 P. Suppes. Popper's analysis of probability in quantum mechanics. (In P. A. Schilpp (Ed.), *The philosophy of Karl Popper*. Vol. 2. La Salle, Ill.: Open Court, 1974. Pp.760-774.)

233 P. Suppes. The promise of universal higher education. (In S. Hook, P. Kurtz, and M. Todorovich (Eds.), *The idea of a modern university*. Buffalo, N. Y.: Prometheus Books, 1974. Pp. 21-32.)

234 P. Suppes. Cognition: A survey. (In J. A. Swets and L. L. Elliott (Eds.), *Psychology and the handicapped child*. Washington, D. C.: U. S. Government Printing Office, 1974.)

235 P. Suppes. The place of theory in educational research. (*Educational Researcher*, 1974, *3* (6), 3-10.)

236 V. R. Charrow. Deaf English--An investigation of the written English competence of deaf adolescents. September 30, 1974.

237 R. C. Atkinson and M. R. Raugh. An application of the mnemonic keyword method to the acquisition of a Russian vocabulary. October 4, 1974.

238 R. L. Smith, N. W. Smith, and F. L. Rawson. CONSTRUCT: In search of a theory of meaning. October 25, 1974.

239 A. Goldberg and P. Suppes. Computer-assisted instruction in elementary logic at the university level. November 8, 1974.

240 R. C. Atkinson. Adaptive instructional systems: Some attempts to optimize the learning process. November 20, 1974.

241 P. Suppes and W. Rottmayer. Automata. (In E. C. Carterette and M. P. Friedman (Eds.), *Handbook of perception*. Vol. 1. *Historical and philosophical roots of perception*. New York: Academic Press, 1974.)

242 P. Suppes. The essential but implicit role of modal concepts in science. (In R. S. Cohen and M. W. Wartofsky (Eds.), *Boston studies in the philosophy of science*, Vol. 20, K. F. Schaffner and R. S. Cohen (Eds.), *PSA* 1972, *Proceedings of the* 1972 *biennial meeting of the Philosophy of Science Association*, Synthese Library, Vol. 64. Dordrecht: Reidel, 1974.)

243 P. Suppes, M. Léveillé, and R. L. Smith. Developmental models of a child's French syntax. December 4, 1974.

244 R. L. Breiger, S. A. Boorman, and P. Arabie. An algorithm for blocking relational data, with applications to social network analysis and comparison with multidimensional scaling. December 13, 1974.

245 P. Suppes. Aristotle's concept of matter and its relation to modern concepts of matter. (*Synthese*, 1974 *28* 27-50.)

246 P. Suppes. The axiomatic method in the empirical sciences. (In L. Henkin et al. (Eds.), *Proceedings of the Tarski symposium, Proceedings of symposia in pure mathematics*, 25. Providence, R. I.: American Mathematical Society, 1974.)

247 P. Suppes. The measurement of belief. (*Journal of the Royal Statistical Society*, Series B, 1974 *36*, 160.)

248 R. Smith. TENEX SAIL. January 10, 1975

249 J. O. Campbell, E. J. Lindsay, and R. C. Atkinson. Predicting reading achievement from measures available during computer-assisted instruction. January 20, 1975.

250 S. A. Weyer and A. B. Cannara. Children learning computer programming: Experiments with languages, curricula and programmable devices. January 27, 1975.